

PROCEEDINGS MICRO-DELCON

THE
DELAWARE BAY
MICROCOMPUTER
CONFERENCE
1978



Sponsored by
IEEE
Computer Society
Delaware Bay
Section



INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS

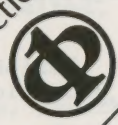


PROCEEDINGS MICRO-DELCON

THE
DELAWARE BAY
MICROCOMPUTER
CONFERENCE
1978



Sponsored by
IEEE
Computer Society
Delaware Bay
Section



INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS



M I C R O - D E L C O N

THE

DELAWARE BAY

MICROCOMPUTER CONFERENCE

MARCH 20, 1978

JOHN M. CLAYTON HALL
UNIVERSITY OF DELAWARE
NEWARK, DELAWARE 19711

Conference Committee Chairman

Conference Chairman
Thomas A. Drake
E. I. du Pont

Publicity Chairman
Charles R. Berg
E. I. du Pont

Program Chairman
David M. Robinson
University of Delaware

Arrangements Chairman
Andrew Zetlan
Delmarva Power

Exhibits Chairman
F. Garey De Angelis
E. I. du Pont

Publications Chairman
Harry Hayman
IEEE Computer Society

KEYNOTE ADDRESS:

Philip R. Thomas
Division Vice President, RCA
"Integrated Circuits - Their
Impact on Society Past, Present,
and Future"

EVENING SPEAKER:

Captain Grace Hopper
Navy Data Automation Command
Department of the Navy

Vendor Exhibits

Digital Equipment Corporation - LSI 11
IBM-5100
Motorola
Peripheral Sales-PHD Computer
Biomation-Eastern Instrumentation
Computerland
Intel
W.F. Keegan & Company
Hewlett Packard Company

TABLE OF CONTENTS

SESSION I HOW TO GET STARTED	
Chairman: James H. Anderson, E.I. du Pont	
"Microcomputer Fundamentals"	2
Joseph M. McKenzie, Western Electric Co., Princeton, New Jersey	
"Designing Logic With Software"	7
A. Scott McPhillips, Information Development & Applications, Inc. Beltsville, Maryland	
"Perspectives on Managing Microcomputer Applications: The Underside of the Iceberg"	12
Henry P. Morneau, E.I. du Pont de Nemours & Co., Wilmington, DE	
SESSION II SYSTEMS ORGANIZATION AND NEW ARCHITECTURE	
Chairman: Louise H. Jones, E. I. du Pont	
"Architecture of the Intel 8086"	15
Steve Kay, INTEL Corporation, Timonium, Maryland	
"CDP 1802 COSMAC Microprocessor-Based Programmable Video Games"	16
A. W. Young, RCA Solid State Division, Somerville, New Jersey	
SESSION III MICROCOMPUTER DEVELOPMENT TOOLS	
Chairman: Charles Wright, Moravian College	
"A Look at the Future of Software Development Tools in the Light of Current Trends"	22
Richard E. Marsh, Hewlett-Packard Company, Avondale, Penna.	
"Optimized Design Cycle for a Micro-Computer Laboratory"	26
David M. Robinson, University of Delaware, Newark, Delaware	
"A Software Development System"	32
R. Hammond, W. Sincoskie, and R. Minnich Univeristy of Delaware, Newark, Delaware	
SESSION IV MICROCOMPUTERS IN BUSINESS	
Chairman: Lester I. Morganstein, E.I. du Pont	
"Information Processing Needs for Small Businesses - A Survey"	38
Robert E. Gibson, University of Delaware, Newark, Delaware	
"Trends in Word Processing"	43
Thomas J. Shinal, Computer Products Unlimited, Inc. Gaithersburg, Maryland	
"A Modular Office System"	48
David A. Farber, University of Delaware, Newark, Delaware Stephen H. Caine, Caine, Farber & Gordon, Inc., Pasadena, California	

TABLE OF CONTENTS (Cont.)

SESSION V CASE STUDIES

Co-Chairman: Joseph Bond, E.I. duPont Co.
John DeGood, Hewlett-Packard Co.

"The Design of a Small Interactive Computer - The IBM 5100" D. A. Roberson, IBM General Systems Division, Boca Raton, Florida	52
"An Approach to Microcomputing with Bit-Slice Microprocessor" Richard J. Smith, Steven L. Clapper, RCA, Missile and Surface Radar, Moorestown, New Jersey.	58
"IMCS-Intelligent Motion Control System" R. A. Patterson, E. I. DuPont de Nemours & Co., Wilmington, Delaware	63
"Data-Linked Bar Code Reader for Documented Manufacture of Health-Sensitive Products" C. Leber, J. Meli, M. Daddazio and R. Jefferis Widener College, Chester, Pennsylvania	65
"Requirements and Benefits of Communications System Standards For Users of Distributed Industrial Control Systems" R. S. Crowder, E. I. du Pont de Nemours & Co., Wilmington, DE (Presentation Only)	

FORWARD

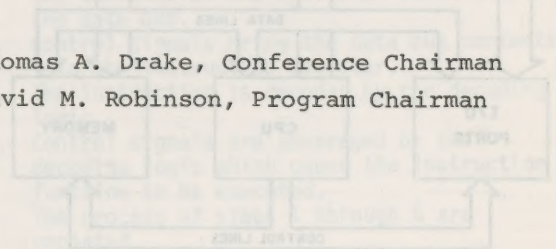
The papers that follow constitute the Proceedings of MICRO - DELCON, a conference addressing microcomputer based systems and sponsored by the COMPUTER SOCIETY CHAPTER of the DELAWARE BAY SECTION of the I.E.E.E. This conference was planned to provide a parochial mechanism for technological interchange among industrial and educational institutions in the Delaware Valley. While the geographical area represented by the participating authors has expanded, the character of locality of reference has been retained.

We especially acknowledge the participation of our invited speakers. Philip R. Thomas, Division Vice President, RCA, delivered the Keynote Address, "INTEGRATED CIRCUITS - THEIR IMPACT ON SOCIETY PAST, PRESENT, AND FUTURE". The Honorable Pierre S. Du Pont, Governor of Delaware, welcomed the attendees during Luncheon and Captain Grace Hopper, Navy Data Automation Command, Department of the Navy, gracefully accepted the role of evening speaker. We thank the SPERRY UNIVAC CO., for sponsoring Captain Hopper's participation.

To the Authors and Session Chairman, we express our gratitude. The technical content of this conference series as it matures will have been well established by your initial efforts.

We appreciate the participation of our exhibitors. The opportunity for attendees to examine and discuss state-of-the-art equipment and devices adds immeasurably to the achievement of the intent of this conference.

Thomas A. Drake, Conference Chairman
David M. Robinson, Program Chairman



MICROCOMPUTER FUNDAMENTALS

Joseph M. McKenzie

Western Electric Company, Princeton, New Jersey

ABSTRACT

In 1971 a complete central processing unit was first incorporated within a single integrated circuit known as a microprocessor. Since that time rapid development of microprocessor capabilities together with dramatic reduction in microprocessor costs has led to the increasing application of microcomputers based on microprocessors and to a vast potential for future application. A typical microcomputer combines a microprocessor with semiconductor memory and input-output ports for external connection. The microprocessor, which is linked to the other units by a data bus, an address bus, and control lines, includes a program counter, instruction decoding logic, an accumulator with an associated arithmetic logic unit, and several general purpose registers. In operation the microprocessor sequentially executes programmed instructions stored in memory.

The CPU is essentially the brain of the computer. It does the arithmetic and logical functions such as adding, subtracting, anding, oring, etc. It must fetch instructions from memory, decode them and execute them. It must cause the entire computer to operate in an orderly and coherent manner via control signals.

Memory is an ordered set of one word storage locations each with a unique address. A word is a group of bits (binary digits) handled by the computer as the basic unit of information. Word lengths vary for different computers but 4 and 8 bits are typical for microcomputers. Memory is used to store the necessary information for proper computer operation. It will hold the program to be run plus the data needed for proper execution of the program. A program is a set of instructions logically linked to perform a particular task or tasks. An instruction, to the computer, is a word whose bit pattern represents a valid code to the CPU instruction decoding logic.

GENERAL COMPUTER ARCHITECTURE

Any computer consists basically of three major elements; a CPU (Central Processing Unit), Memory and I/O (Input/Output) Ports. See Figure 1.

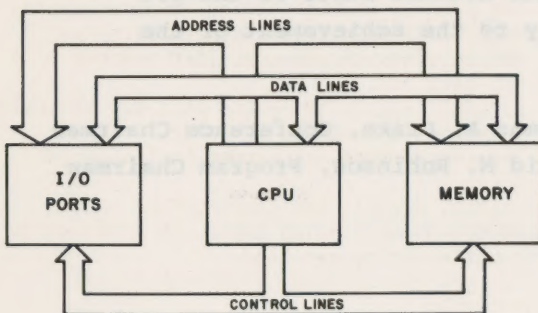


FIGURE 1
GENERAL COMPUTER ARCHITECTURE

I/O ports allow the computer to communicate with the outside world. This would include printing devices, displays, keyboards and switches. Thus a computer can transmit its "answers" to human beings via displays and/or printing devices and human beings can transmit input parameters to the computer via a keyboard or switches.

These three elements (CPU, Memory and I/O Ports) must be interconnected by three basic sets of signals; data lines, address lines and control lines. These lines are generally referred to as buses. The data lines carry the data words to and from memory. These data could be instructions, numbers, logical bit patterns or codes representing printable characters. The address lines are used to access the proper memory location or I/O port. Finally the control lines are used to coordinate the entire computer function to insure a logical and coherent operation. These consist of such signals as a Memory Read Pulse to indicate to memory that a read operation is to be performed or an I/O Output Pulse to indicate an I/O output. Similar pulses are available for the other I/O and memory functions.

HISTORY

In 1971 Intel Corporation marketed the first commercially available microprocessor, the 4004. It was a 4 bit word device, packaged in a 16 pin DIP (Dual Inline Package) and had a total of 46 instructions in its repertoire. It was much slower (10-20 microseconds per instruction) than the present day minicomputers but it did represent a technological breakthrough by incorporating in a single IC (Integrated Circuit) the CPU portion of a computer.

This was only the beginning. The next year Intel marketed the 8008 microprocessor. This was an 8 bit device which was not much faster in execution than the 4004 but it had the important interrupt capability. Interrupt capability allows a CPU to respond to an outside signal by temporarily suspending operation of its regular programming to service the interrupting device.

In 1973 the Intel 8080 microprocessor hit the market. This was a huge improvement over previous processors. It was an 8 bit device with instruction execution speed of 2-9 microseconds. It had interrupt capability, was housed in a 40 pin DIP, which allowed 64K of memory to be addressed and had a set of 76 instructions. This CPU has become so popular and widespread that it is considered by many to be an industry standard.

In 1977 Intel produced the 8085 which has the same instruction set as the 8080 but has incorporated into the chip a clock circuit and some system control circuitry. This reduced the total number of system IC's required since both clock and system control circuits were needed with the 8080 device.

Late in 1977 a new trend appeared. The technology had reached the point where not only CPU functions but memory and I/O ports could be integrated into one silicon chip. So now we have available, or shortly to come, complete computers in one IC. The typical configuration consists of the CPU, 1K or 2K of ROM (Read Only Memory), 64 to 128 words of RAM (Read/Write Memory) and two or three I/O Ports.

MICROPROCESSOR ARCHITECTURE

A typical microprocessor configuration is shown in Figure 2. Basically a microprocessor contains the following major elements:

1. Arithmetic Logic Unit (ALU)
2. Accumulator
3. Instruction Register
4. Instruction Decoding Logic
5. Program Counter
6. General Purpose Registers
7. Control and Timing Logic

The ALU is the circuitry that performs the addition, subtraction, anding, oring and similar arithmetic and logical operations required by the computer. The accumulator is the major register (on chip temporary storage location) in the computer. It serves as one of the inputs to the ALU and receives the results of most ALU operations. It is the work horse register of the computer. The instruction register is the register that receives the instruction word from memory. The instruction decoding logic is circuitry which examines the bit pattern in the instruction register and accordingly generates the appropriate control signals to implement the instruction function. The program counter is a register which holds the address of the instruction to be executed. General purpose registers can be used as temporary storage locations for data. These registers also can be arithmetically and logically manipulated under instruction (program) control. Finally, the control and timing logic is circuitry that sequences CPU operations and causes the entire unit to function in an orderly and coherent manner.

As can be seen, the CPU of a computer is no simple entity. It is a huge logical conglomerate operating in concert with the instructions of the program to produce a useful function (controlling a manufacturing process, causing a car engine to run optimally, playing a game on a TV screen, testing product, etc.). And this circuitry is contained in a piece of silicon about two tenths of an inch square!!!!

MICROPROCESSOR INSTRUCTION

Assume a program resides in memory which is properly connected to the CPU. Also assume the program counter has been set to the starting address of the program. Now by the push of a "button" or an input command from some input device (TTY for example), the processor starts to function and the following occurs:

1. The program counter contents are placed on address bus.
2. The program counter is incremented by one.
3. Control signals tell the memory to read the contents of the specified address onto the data bus.
4. Control signals bring the data bus contents into the instruction register.
5. The instruction is decoded by the decoding logic.
6. Control signals are generated by the decoding logic which cause the instruction function to be executed.
7. The process of steps 1 through 6 are repeated.

As can be seen this Fetch, Decode, Execute pattern is repeated over and over. In this way the computer steps through the set of instructions that constitute a program.

It may be worthwhile to examine the execute portion of some typical instructions. Assume the following three instructions are valid for the microprocessor under consideration:

```
ADD R2
MOV R1, R3
JMP ADDRESS
```

The ADD instruction when executed will add the contents of general purpose register R2 (each register has a code name) to the accumulator and put the result in the accumulator. The MOV instruction will move the bit pattern in general purpose register R1 into R3. The JMP instruction will cause the program to jump to whatever address the symbol ADDRESS represents. The symbol or label "ADDRESS" is a code for an actual number which is an actual address in memory.

All instructions are fetched and decoded in the same manner. The execute portion of the cycle is where variation occurs. For the ADD instruction execution occurs when the register R2 is routed onto the internal data bus (See Figure 2) and into the ALU.

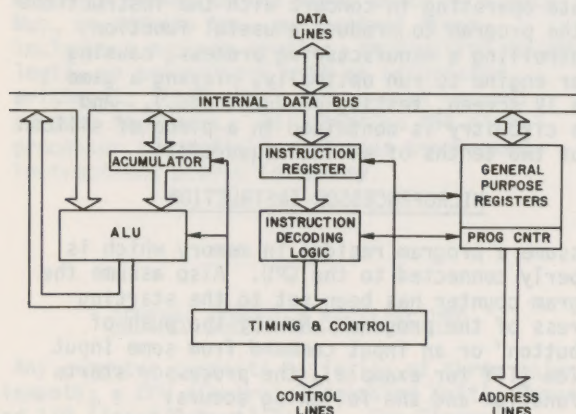


FIGURE 2 - MICROPROCESSOR ARCHITECTURE

Concurrently the accumulator contents enter the ALU and addition takes place. The output of the ALU is then routed to the internal data bus and into the accumulator. Execution of the instruction is completed. For the MOV R1, R3 instruction execution occurs when the contents of register R1 are placed in register R3 via the internal data bus. For the JMP ADDRESS instruction execution occurs when the number (or bit pattern) represented by the code ADDRESS is placed into the program counter so the next instruction fetched will be from the memory location "ADDRESS". The JMP instruction requires an address which must be in memory immediately following the JMP code or bit pattern. If we assume a microprocessor with addressing capability of 64K it will take two 8 bit words to fully define the address. Therefore a JMP instruction becomes a three word instruction. One word is the code for the JMP

instruction and the next two words are the address. Thus when the JMP instruction is decoded it tells the processor that two more memory reads must be done to complete the instruction. When these two words are brought in over the data bus they are temporarily stored in two temporary storage registers not shown in Figure 2. When the full address has been read into the temporary storage registers, the instruction can be finally executed by transferring the address into the program counter then proceeding to the next instruction by initiating a fetch cycle.

Subroutines are an important part of a computer's operation. A subroutine is a set of instructions intended to be used many times in a particular program. A subroutine can be called from any place in a main line program and this is accomplished by a jump to subroutine (JSR) instruction. Obviously the JSR must be accompanied by an address (or code of an address) which indicates where in memory the subroutine begins. For instance, a JSR TTYIN instruction would call a subroutine which starts at address TTYIN (code for an actual address) and may be a routine to input a character from a Teletype (TTY). Now, since the intent when calling a subroutine is to temporarily leave the main line program and return to the exact point of exit, the program counter must be temporarily stored just before the program jumps to the subroutine and retrieved when the subroutine is completed. The JSR instruction automatically stores the program counter in a special place in memory called the stack. A return (RET) instruction, which must be the last instruction in any subroutine, will automatically retrieve this stored address from the stack and place it in the program counter. Thusly continuity of the main line program is preserved.

An important feature of most microprocessors is the stack and associated stack pointer. A stack is an implicitly designated area in memory used for temporary storage. The particular address being accessed during a stack storage operation is determined by a special CPU register called the stack pointer (SP). The stack pointer register must be loaded by a load stack pointer (LDSP) instruction to insure that the proper area of memory is used for the stack without interference with program and data. Writing onto the stack is usually called a PUSH and reading from the stack is called a POP. An example of how the stack concept operates is best shown by a subroutine call. Before a subroutine is called the stack pointer is loaded with a LDSP ADDRESS instruction. When a JSR instruction is encountered the program counter is automatically PUSHED onto the stack and then the stack pointer is decremented. In this way the stack is ready for another PUSH if need be without destroying the previously saved data. When the subroutine is finished and a RET instruction executed the stack pointer is first incremented and then the memory contents pointed to by the stack pointer are

POPPED off the stack and into the program counter. With this last in first out (LIFO) operation of the stack it can be easily seen how subroutines can be nested (that is one subroutine call another subroutine) and the continuity of the overall program flow be preserved.

MEMORY

The preferred type of memory for use in microcomputers is semiconductor memory. Nothing precludes the use of ferrite core memory, but semiconductor memory is almost always the choice. Semiconductor memories fall into two basic categories, Read/Write Memory (referred to in the industry as RAM) and Read Only Memory (ROM). As the name implies Read/Write Memory (RAM) can be read from and written into, hence can be used for temporary data storage and retrieval. Read Only Memory (ROM), on the other hand, has permanent contents which can be retrieved but not altered. The contents of ROM are determined and fixed prior to its incorporation into the microcomputer architecture. One outstanding characteristic of ROM memory is that it is non-volatile. A non-volatile memory is a memory that retains its contents when power is removed and restored. RAM memory is volatile and loses its contents when power is removed. Restoration of power will result in random contents.

ROM memories can be categorized broadly as mask made, programmable and re-programmable. The mask made ROMs have contents that are fixed at the time of manufacture. These are permanent contents and cannot be changed. The programmable ROMs are manufactured with "1's" in every memory location. Part of the manufacturing process includes the incorporation of fuse-like connections to all memory cells (one bit of memory). Special equipment is needed to allow the selective "blowing" of these "fuses" to obtain the desired pattern of "0's" and "1's" in the memory structure. The re-programmable ROMs (call Re-PROMs, EPROMs or merely PROMs) can be programmed using special circuits designed for this purpose but can also be erased to all "0's" by exposing the chip to ultraviolet light for a specified time (10-15 minutes typical). After erasure they can be reprogrammed thus providing great flexibility in use.

Many applications of microcomputers require both ROM and RAM. The basic program would reside in ROM which would give it protection from any power failure or shutdown. RAM would be used for temporary data storage (stack, for example).

I/O DEVICE INTERFACING

I/O ports allow the microcomputer to communicate with the outside world (TTY, Printers, Displays, etc.). To provide the necessary drive for output devices and to coordinate the reception of data from input devices, a microprocessor generally requires some type of latching registers (Flip-Flops) to serve as I/O ports. These types of latches are readily available in a single IC

chip.

Interfacing a microprocessor to I/O devices presents the same problem as interfacing a mini. Transducers of some kind are needed for input if the stimulus is non-electrical such as pressure, temperature, etc. Once the signal becomes electrical in nature it must then be converted to a form compatible with the microprocessor being used. This usually means a parallel 8 bit digital word. Analog to digital (A to D) converters for this purpose are readily available on the market. Also serial to parallel data conversion circuitry in a single IC chip is a common off the shelf item (UARTs for example for use with TTYS, CRTs, etc.). Of course in many cases the input is already in digital form (logic data, relay closures, any on/off type signals) and this can greatly simplify the interface problem.

For output signals the data translation requirements are similar, merely the direction of data flow is changed. Thus, for example, digital to analog (D to A) would be required in some cases. However, many outputs are displays and these are often compatible with computer parallel type digital data.

In summary, the advent of the microprocessor has provided no new solutions to the ever present interfacing problem. However, it must be added that microcomputer manufacturers are producing IC's that address the interfacing problem and tend to make life easier for the design engineer.

USE OF MICROCOMPUTERS

One of the most appropriate uses of a microcomputer is for logic design. Before the advent of the microcomputer, logic design was accomplished by connecting many IC's (NAND gates, NOR gates, inverters, etc.) in the proper configuration to produce the desired function. Debugging of the design usually required much time and effort. Also, once a design was fixed and implemented in the field, any change was a very expensive and tedious undertaking. Microcomputers have changed this. A designer now programs his function instead of hard wiring huge arrays of gates. Debugging becomes much faster and easier (with the proper software debugging aids). Change of design after implementation becomes merely a program change instead of expensive hardware redesign. In fact, several functions can be programmed into ROMs and changes from one function to another accomplished by merely inserting the proper ROMs in the proper sockets. Thus flexibility of design is a real advantage in using microcomputers in logic design.

Another advantage of the microcomputer approach to logic design is greater reliability. A microcomputer generally will require fewer ICs than the equivalent hardware logic design. This means less interconnections which means increased reliability since interconnections are the source

of most failures in electronic design. The smaller size due to the fewer IC's can also be a big advantage in certain cases such as airborne equipment.

In view of the above it becomes apparent that microcomputers can result in reduced logic design and manufacturing costs. This economic advantage plus the attendant increased reliability and flexibility will destine the microcomputer to become paramount in electronic design applications.

DESIGNING LOGIC WITH SOFTWARE

A. Scott McPhillips

Information Development & Applications, Inc.
Beltsville, Maryland

Microprocessor software is the latest "component" to be applied to digital system design. It is well suited to implementing complex logic like controllers and interactive operator panels. Present design notations, such as logic diagrams or program flowcharts, do not adapt well to logic design with software. A new methodology combines top-down structured programming concepts with state diagrams or tables. The resulting design notation is capable of concisely describing both application requirements and a corresponding program. Experience with the technique has shown that it leads to reliable programs that are both compact and efficient.

The Problem

Microcomputers are widely heralded as replacements for logic circuitry, at least in applications that can stand a little delay in the logic. However, microcomputers are certainly not a direct replacement in the sense that IC's can replace transistor or relay circuitry. Instead, effective utilization of microcomputers in logic applications requires a complete rethinking of the problem, as well as acquiring new knowledge, skills, tools and perhaps people. Microcomputers also force one to deal with one of the most expensive and notoriously unreliable creations of technology: software. These facts give rise to an important question. Why bother? Will a microcomputer be beneficial enough to be worth the trouble?

Among all the possible reasons that might be cited for using microcomputers one emerges that is truly profound in its impact on both designs and designers. With a microcomputer, a complex, sophisticated design can be built with exactly the same parts as a simple design. This means that extra capability--and lots of it--can be added to a design without having to pay a price in reliability, size or maintainability. The microcomputer actually combines the virtues of

simplicity (in the hardware) with the performance attainable only with complexity (in the software).

The impact of this dual personality of the microcomputer can be seen in many of today's new products. For example, instead of displaying a simple sensor reading a microcomputer can display an average value with a guaranteed standard deviation. A microcomputer controlled telephone system can easily provide automatic call forwarding and abbreviated dialing. In an industrial control system a microcomputer can automatically adjust gains and delays as circumstances change and it can provide an automatic start-up sequence as well. These kinds of features add enormously to a design's value but, with a microcomputer, they cost almost nothing.

Thus the microcomputer's abilities encourage designers to create very complex and sophisticated machines. The problem, of course, is that it is difficult to design complex and sophisticated machines well. Complexity often overwhelms us with details, leading to designs with bugs because some obscure combination of events was overlooked by the designer.

The only solution to this problem is meticulous design, aided by reviews and cross-checks to isolate mistakes. As a vehicle to encourage this kind of designing we need methods and notations that help to manage complexity. There are many widely used aids for managing hardware designs; schematics, ladder diagrams, and logic diagrams are a few examples. Very few such aids exist for software. This paper describes one developed specifically for implementing logic with microcomputer software.

The FSM Approach

Logic design has been practiced with gears, relays, tubes, transistors and even fluidics. Its principles remain the same with any technology because components merely implement in

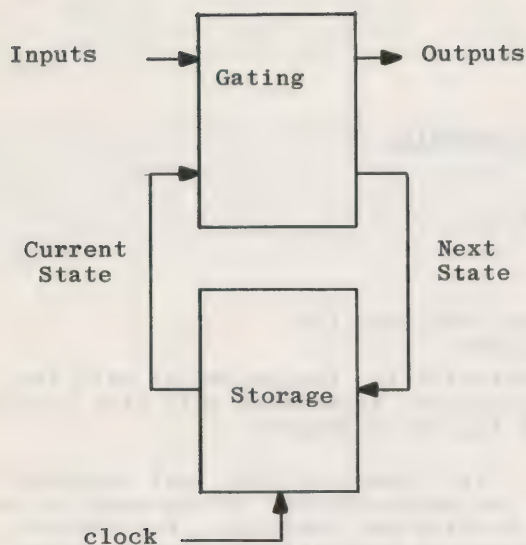


Figure 1 An abstract view of a finite state machine implemented with hardware.

some physical way the fundamental concepts of boolean algebra. In order to apply these universal principles to the design of microcomputer software, it is helpful to start with fundamentals and acquire an abstract, technology-independent view of the machines we wish to build.

The finite state machine (FSM) concept is ideal for this purpose because it can be applied to any digital system. Figure 1 shows a common way of visualizing an FSM as two kinds of elements, storage and gating. Taken as a whole, the content of all storage elements in a system represents the system's present state. Logic gates combine the state with the input signals in some manner to create the system's next state.

It is, unfortunately, not possible to directly translate a TTL logic design into software. However, it is eminently possible to translate an abstract FSM design into either a TTL design or a software design. In fact, it is much easier with software than with hardware! Thus, a possible design methodology appears because any digital system can be described as an FSM.

If it is indeed easy to implement FSMs with software, then a rational approach to designing logic in software exists. A system to be designed can first be specified as an FSM, beginning perhaps as sketches like Figure 2.

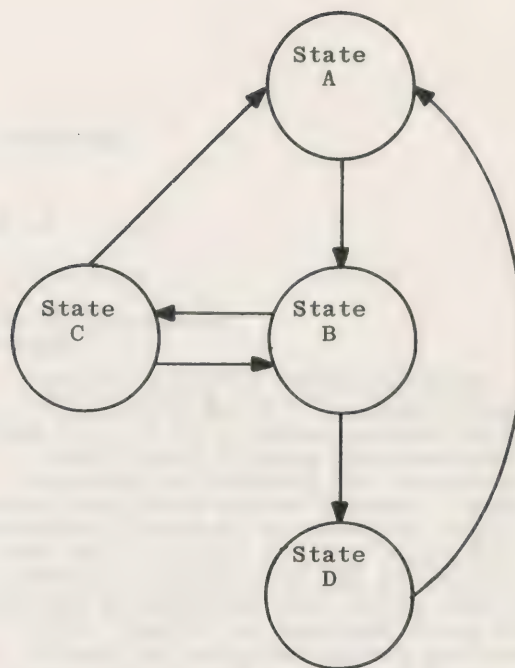


Figure 2 An abstract view of a finite state machine implemented with software.

The system's states are identified and cataloged along with the events or conditions that should cause each state change. This information comprises a fairly rigorous specification for the software. Specifications are a rare thing in the software world but an absolute necessity for good design.

The process of developing an FSM description can be difficult and time consuming but it is the essence of creating a design. Fortunately it is not a very error prone operation because the rigorous structure of FSMs forces one to consider all possibilities. By dividing the problem into states, the FSM approach introduces a convenient way to divide and conquer, permitting the designer to concentrate on small, well-defined portions of the design without confusion. By considering a list of all system inputs every time a new state is examined, one can reliably isolate and define the desired system behavior under all circumstances.

Design Example

Figures 3 and 4 illustrate useful techniques for applying the FSM concept. The design is a simple motor controller that operates at one of two speeds. The motor has a brake for stopping and a special winding for starting. In addition, an RPM readout is to display actual speed while accelerating, but will display the fixed setpoint when the servo is in control.

The overall logic can be clearly illustrated by a state diagram such as Figure 3. After a few iterations, one

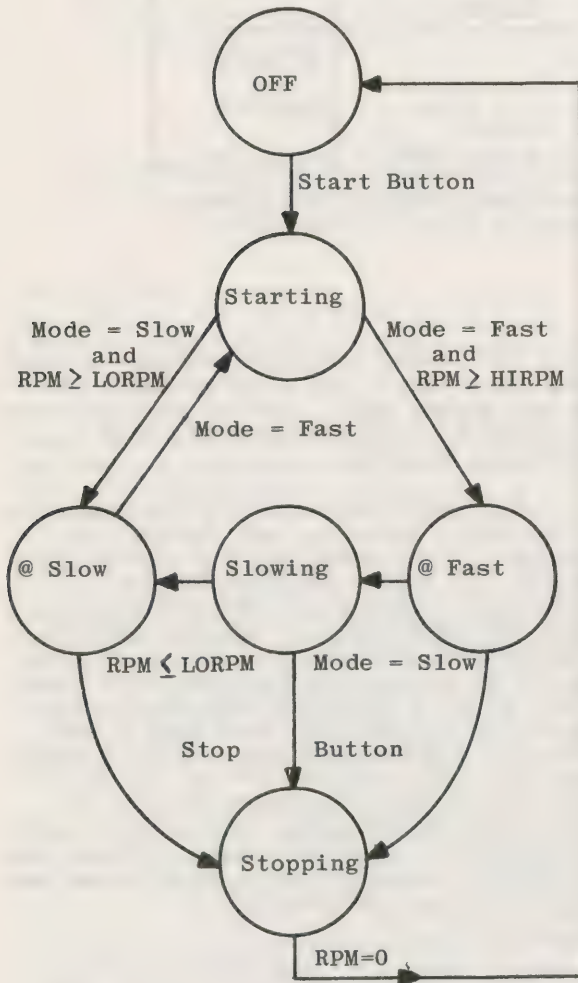


Figure 3 State transition diagram for the motor controller design example.

can develop a fairly simple looking diagram that succinctly names the system's states and graphically illustrates the major characteristics of the design. In Figure 3, for example, it is easy to follow how the motor gets started and eventually reaches either the "@ slow" or "@ fast" state. Additional logic shows what happens if the mode changes between fast and slow while running.

A diagram like Figure 3 may be adequate for describing a simple design but it soon becomes cumbersome in most applications. A more practical way to represent FSMs is shown in Figure 4. This tabular approach readily expands to include any number of states and transitions. For each system state, all events of interest in that state are listed. For each event, the corresponding action to be taken is described, along with the next state to be entered. The terms and language used within the state table can be developed along with the table. The intent should be to clearly indicate what is to be done, not how it will be accomplished. Thus the table may well include simple statements, conditional checks, formulas, some programming language, anything as long as it is clear.

In a complex application the process of creating the state table is intricate and lengthy. None of it is wasted effort though, abstract as it may seem. The completed table is analogous to a detailed block diagram of a hardware design. It identifies all the parts of the design, their functions and their interactions. Like a block diagram, it also serves admirably as a representation of the proposed design that can be used to review the design with colleagues, marketing people and customers. The state table clearly answers "what will happen if ...?" questions and it is easy to modify if some of the answers come up unacceptable.

Writing The Program

Programming has a bad reputation with many engineers because it is often done poorly, heuristically, and by amateurs. On the other hand, there are some first-class program designers from whom we can learn much. Observation of the working habits of good professional programmers has shown that they spend their time approximately like this:

Design:	60%
Coding:	20%
Debugging:	20%

State	Event	Action	Next State
Off	Start Button	Lights = On + Start Brake = Off Display = Actual RPM Triac = Start Voltage	Starting
Starting	Stop Button	Lights = Brake Brake = On Display = Actual RPM Triac = Off	Stopping
	Mode = Fast and RPM $\uparrow$ HI	Lights = On + High Triac = HI Servo Display = HI RPM	@ Fast
	Mode = Slow and RPM $\downarrow$ LO	Lights = ON + Low Triacs = LO Servo Display = LO RPM	@ Slow
@ Fast	Stop Button	as above	Stopping
	Mode = Slow	Lights = ON + Slow Triacs = LOSERVO Display = Actual RPM	Slowing

Figure 4 A state table shows much more detail than diagrams can. This sample describes three of the motor controller states.

Even more important than the exact numbers is what the choice of categories implies. The program is designed before it is coded in a computer language. If we are to succeed in emulating the performance of the best programmers then we should thoroughly design a program before writing it. To do otherwise would be like trying to skip the logic diagram in a hardware design. Few of us can design circuitry directly on a PC layout or wire wrapping list. Instead we create a paper representation of the design--the logic diagram--and then handle the detail work of implementing it with components.

This is why the state table was developed. It is a very detailed and concise paper representation of the design. It is applicable to any micro-computer or software language, just as a logic diagram can be implemented with any of several component families and in many packaging configurations. Completing the state table doesn't mark the beginning of the programming. Rather, it means that the programming is more than half done!

What remains to be done is a process of translation. The state table is really a program that is written in a language we like because it fits the problem well. It must now be converted

into the chosen computer language. This task takes skill and craftsmanship but it is not design work. Coding is a lot like creating a working breadboard from a schematic diagram.

The important contribution that the state table makes is that during the program coding phase one can concentrate solely on the translation problems. A good state table will separate design from implementation so sharply that an experienced designer will often be able to sit at a terminal and type in a first draft program directly from the table. Those operations that need only a few computer instructions are typed in immediately; more complex work is handled by calling a not-yet-written sub-routine. The resulting program, after a few editing sessions, will be well organized and easy to follow because it will clearly reflect the conceptual FSM design.

The flowchart of a program that models an FSM will generally look like Figure 5. There is often logic to be performed independent of the system's state. This is handled first in the "global Logic" section of the diagram.

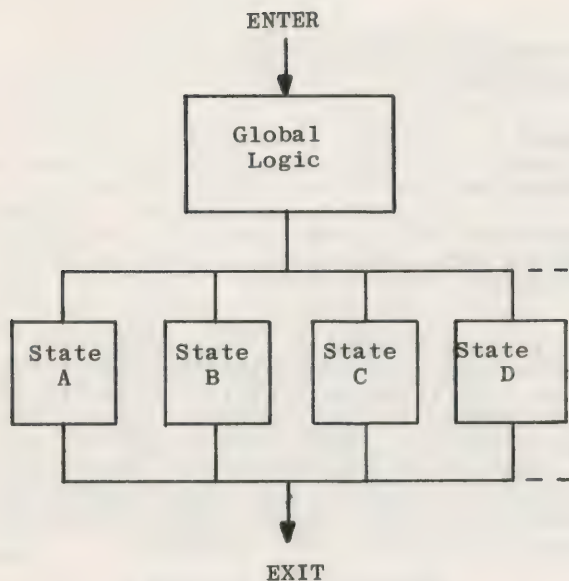


Figure 5 Typical flowchart for a program that implements a finite state machine.

In the motor controller, for example, the stop button could be handled here since it does the same thing in all states.

The rest of the program consists of many small programs, one for each state. The state is represented by a number stored in the computer's memory. The program examines the state and branches to the correct section. The currently active state's program then tests for all of the events listed in the state table for its state. If one of these events has occurred then the state program reacts taking the appropriate actions and updating the value of the state. The next time the program is executed it once again performs the global logic and then branches to the new state's program.

The model flowchart also illustrates another important characteristic. When the FSM program is entered it performs a few tasks and then exits. It is usually important to arrange the program in this way so that it can share the microcomputer with other programs. In a typical application the microcomputer could control a number of devices, perhaps a motor plus a heater, a few valves, a keyboard and display and a small data acquisition system. If each of these subsystem programs are self-contained and structured with a single entrance and exit, then it is a simple matter to string them into a chain, thus implementing several

FSMs more or less simultaneously on a single microcomputer.

Conclusions

Programs designed using the FSM approach have many virtues. The formality and structured nature of the technique help to avoid oversights and confusion. The states break the program into many small and easily understood pieces with very little intercommunication. This makes the program easy to understand and modify.

Perhaps surprisingly, an FSM program is also very efficient. It retains everything it needs to know about the past in just one variable: the current state. In contrast, a typical program designed with ad hoc techniques will retain information about the past in numerous variables (flags) that indicate something has happened. This kind of program not only consumes memory for the flags, it consumes even more memory with the programming that sets and tests them. Instead of testing flags to decide what to do, an FSM program looks up its state and then does exactly what is needed, no more and no less. Thus organizing a program around the FSM concept usually produces both the smallest and fastest program design.

Experience with the technique has shown it to be useful in almost any microcomputer application. It provides a clear and convenient representation of a program design before the program is written. It encourages and facilitates thorough review of the design at a stage when it is economical to make changes. Finally, it leads directly to efficient programs that are reliable and maintainable because they were designed, not just written.

PERSPECTIVES ON MANAGING MICROCOMPUTER APPLICATIONS:
THE UNDERSIDE OF THE ICEBERG

Henry P. Morneau
Senior Supervisor
Engineering Physics Lab

Engineering Department
E. I. du Pont de Nemours & Co.
Wilmington, Delaware

ABSTRACT

The dynamic environment created by the rapidly growing technology of microcomputers places many demands on everyone associated with their application. This paper focuses on some recent experiences in organizing, planning, controlling, and directing a technical staff involved in the development and design of microcomputers for industrial control systems.

INTRODUCTION

Rapid growth in computer technology during the past 25 years and more recently the explosion in microcomputer technology, coupled with the demands of our society to meet an expanding economy, has created a fertile environment for the application of this technology. One doesn't have to look far to see the diverse areas where micros are already being used - home appliances, electronic games, medical and industrial instrumentation, automotive and industrial controls, etc. And it is unusual to find technical journals and conferences that aren't focusing their efforts on some aspect of computer technology. Practically everywhere you turn, micros are in the limelight.

Because of this fertile environment, because there are great opportunities available, because of the recognition and glamour associated with this field of work, and because of our appetite for technical challenges, a large number of people have become involved and perhaps a larger number of people want to become involved in this technology. This paper is directed at those who have little or no experience in this field, but have a yearning to get involved. It highlights some of the key lessons I've learned during ten years of active participation in the development and application of mini- and microcomputers to industrial control and instrumentation systems. More importantly, it addresses some

of the key obstacles I've encountered while organizing and planning the activities of a technical staff involved in the development and application of microcomputer systems. For those of you who have a yearning to get involved in this technology, it will provide some insight into what lies ahead. This is a brief glimpse beyond the tip of the iceberg!

OBSTACLES TO SUCCESS

Managing microcomputer applications has been challenging, frustrating, enjoyable, and rewarding. In many cases the entire development/application process went as planned, yielding the desired results. In other cases many obstacles were encountered which were not anticipated and prevented us from reaching some or all of our goals. These obstacles, or more appropriately some of the mass beneath the tip of the iceberg, are described below. These items have been divided into three chronological phases associated with the application of this technology - selling, implementation, and support.

Selling

Very few successful businessmen will agree to incorporate micros into their products just because it represents a new and exciting technology. There must be an economic incentive! In establishing this incentive, all of the cost and savings factors must be considered. Experience has shown that identifying all of these economic factors is no easy chore and that quantifying them is even more difficult. For example, we all recognize the difficulty in estimating software costs in any application. When micros are involved, software represents a much higher ratio of the total cost, making it more difficult to arrive at a reasonably accurate estimate. At the same time, savings resulting from decreased maintenance are very difficult to quantify. Even more difficult is our ability to quantify the value of a system feature. For example, how much is the consumer willing to pay for an oven

controller that ramps the temperature during the cooking cycle?

Once you've established a convincing economic picture, there are other obstacles which must be overcome. As with any investment, and that is how a client will be looking at the effort required to incorporate a micro in his product, there is an element of risk which must be considered. Your client, whether it be your management, other departments within your company, or other corporations, has surely heard a number of horror stories describing other efforts to apply computers and microcomputers. Your client will, and should, question the capabilities and experience that can be applied to his product. He will be looking for assurance that you have the necessary technical abilities, but more importantly he will be looking at the way you manage your business for assurance that the work can be completed on time and within estimated cost.

One other aspect of selling that is important is the tendency to oversell. It is easy to get carried away in describing all of the capabilities of the micro when compared to the conventional methods being used without discussing the cost necessary to implement them. Then, once the decision has been made to proceed with the application, the client begins to ask for these capabilities which he has been told are possible. All too often, these features have not been included in the economic analysis. This can generate bad feelings and lack of confidence, setting the project off to a bad start.

Implementation

One of the most important decisions to be made early in the implementation phase is the selection of microcomputer hardware and software. From a management point of view a decision has to be made; and, while the decision may be easy to make, defending that decision can consume a great deal of time and effort. After you've added all the pluses and minuses for several potential models, there may not be a substantial difference between the alternatives. The problem is often reduced to that of debating the pro's and con's of Chevrolet vs. Ford, and don't expect unanimous approval for the decision, either!

Proceeding with software development and system design, experience has shown that the problem being solved tends to get bigger and requires considerably more effort than originally estimated. Some of this is due to the extras that are being added; however, the bulk of it is just our

inability to see beyond the tip of the iceberg until we begin exploring it. There's almost an infinite number of obstacles which arise, some of which are described below.

All of those seemingly minor extras that were added begin to accumulate, requiring more memory, I/O, and other peripherals; the software tasks get larger and more complex, requiring greater effort to meet project timing; the hardware doesn't work quite the same way the vendor described or you find the hardware doesn't have some feature you assumed it had; your lead engineer leaves for greener pastures, gets run over by a truck, or gets pulled off for a more pressing problem, and you don't have an adequate replacement; vendors can't meet required delivery schedules because production can't keep up with the demand or they've discovered a design error and must implement an engineering change before resuming production; software testing and hardware check-out begin competing for the same facilities, necessitating shift work and overtime; inexperience in programming micros leads to numerous design and coding errors which are difficult to debug; insufficient test facilities lengthen the check-out phase beyond your wildest expectations; and, just when you think you're ready for system check-out, you find the product this micro is going to be incorporated into has been redesigned. Of course, it's an innocent error; the product designers just didn't think it would have any impact on the computer!

Follow-Up

The effort doesn't stop when you've finally completed system check-out and the total product performs as intended. Again, experience has shown that as the user becomes acquainted with the product he contributes many worthwhile suggestions that never crossed your mind, or he finds the product cumbersome to use and suggests drastic changes to make it more effective. In addition, through the evolution of the product the design team also has accumulated a list of changes they would like to see implemented. In most cases there is follow-up design producing models 2, 3, 4, and many more. In addition items such as documentation, maintenance, and training, which require considerable effort, must be addressed. Failure to provide adequate follow-up is likely to generate bad feelings, lack of confidence, and little hope for future applications.

COPING WITH OBSTACLES

From a realistic point of view, managing the development and application activities associated with micros isn't much different in form from managing any other technical activity. The numerous obstacles to success which are encountered are not necessarily unique. The prime difference is in the quantitative or dynamic aspects - the amount and frequency of technological change; the amount and frequency of training and developing personnel; the size of the knowledge gap which leads to communication problems; the large number of applications; etc. It's as if 100 years of change has been compressed into 10 years. So in many respects managing in today's microcomputer world requires dealing effectively with change.¹

To deal effectively with change, we have to recognize and admit that it exists, we have to believe that something can be done to control it for our benefit, and we have to accept the frustrations which occur from time to time when things don't go as planned.

Recognizing change is something we all feel is quite easy, but in reality it is very difficult. We have a tendency to become lost in our day-to-day activities and neglect to look to the future. Often we are blinded by our subjective ways and fail to admit that our environment is changing. An important element of managing this business is to spend the time and effort to look ahead, objectively assess what is changing, and chart a course for the future. To investigate, plan, and prepare for the future will require budgeting funds and allocating resources. It requires a commitment that many of us are not willing to make or are not capable of making. Recognizing change and preparing for it are necessary for anyone involved in today's microcomputer world.

Controlling the results of change (assuming that it is unrealistic to control change itself) can go a long way toward reducing the obstacles described earlier. To maintain this control requires an organization that is aggressive, flexible, talented, and highly motivated. Organization is a large subject^{2,3} area that I won't deal with here; however, I do want to make a few remarks relative to motivation.

In my experiences in managing a development team, it has been demonstrated on numerous occasions that a highly motivated person can and will overcome many obstacles which prevent him from reaching his goals. On the other hand, people that

don't have this high degree of motivation give in to the frustrations and challenges and often fail to reach their goals. Working in a changing environment such as micros requires a commitment to stay abreast of the technology, to broaden one's abilities to deal with issues such as economic analysis and selling, and to apply extra effort when the going gets tough. Without this element of motivation and commitment, the organization is at best second-rate and more likely will fail.⁴

Experience has shown, also, that one can't control all of the effects produced by a changing technology. Often global decisions are made which optimize the benefits of the whole while minimizing those of some of the parts. If these decisions are good ones, and that requires an objective analysis, then the best move is to reassess the state of the current and future business in light of these decisions and chart a new course.

If we accept this philosophy of dealing effectively with change, then it is possible to organize personnel and create an environment that not only can deal with it effectively, but is also highly motivated by it.

SUMMARY

Rapid advances in microcomputer technology have had a major impact on our society and have created the need for many people to become involved in its application. Managing the development and application activities often requires a greater effort than originally perceived because many factors which can't be seen at the outset impose obstacles to successful completion of project goals. Recognizing the obstacles, why they exist, and implementing effective management practices to deal with the constantly changing world of microcomputers will increase the probability of success and the satisfaction derived through successful implementation.

BIBLIOGRAPHY

1. Morgan, J.S., "Managing Change," McGraw-Hill and Company, New York, 1972.
2. Gee, E.A. and Tyler, C., "Managing Innovation," John Wiley & Sons, New York, 1976.
3. Drucker, P.F., "The Practice of Management," Harper & Row, New York, 1954.
4. Evans, C.G., "Supervising R&D Personnel," American Management Association, Inc., 1969.

ARCHITECTURE OF THE INTEL 8086

Steve Kay

INTEL CORPORATION
Timonium, Maryland

ABSTRACT

The INTEL 8086 16 Bit processor is previewed. Preliminary part specifications are presented and the instruction set is described. Details of the total system hardware and software support are presented.

(Ed. Note - This paper not available at publishing time)

CDP1802 COSMAC MICROPROCESSOR-BASED PROGRAMMABLE VIDEO GAMES

A.W. Young
RCA Solid State Division
Somerville, N.J. 08876

Abstract

A programmable video-game system based on the features and architecture of the CDP1802 microprocessor is described. The system consists of the microprocessor, a ROM program memory, a RAM scratchpad and display memory, a CRT display controller, a hex keyboard interface, and programmable sound. A key feature of the design is a bit-mapped graphics display with refresh controlled by means of the CDP1802 DMA interface. Other features include sensed keyboard activity through the use of CPU input flags, controlled sound by means of a single-bit CPU output port, and ROM chip enable decoding logic on the ROM chip. Tying the system together is an interpretive game-oriented programming language consisting of 31 two-byte instructions. Instructions are provided to generate random numbers, read hex keyboard inputs, display video patterns, sound a tone, or manipulate variables.

Introduction

Video games represent an application particularly appropriate for microprocessor implementation. Unlike hardwired logic, a microprocessor-based game system can be easily reprogrammed to realize a broad range of game applications. The microprocessor-based system described in this paper, although specifically designed around the COSMAC CDP1802 as a sample application of that device, is general in concept. A block diagram of the system is shown in Fig. 1; it consists of the three standard microcomputer functions of microprocessor, I/O, and memory. The COSMAC CDP1802 microprocessor₁ is discussed in detail elsewhere.

I/O

As shown in Fig. 1, the game system I/O consists of the video display controller, keyboard interface, and sound interface. As with most microcomputer applications, the I/O design personalizes the computer hardware to a specific application.

A key feature of both the I/O hardware and system design is the video-display controller. In the general case, the display controller performs the functions of video generation, sync generation, and refresh.

Video generation - The function of video generation usually follows one of two techniques, bit mapped or character generation. In the bit-mapped display, the technique used in this example, each bit in a display memory maps to one spot on the video screen; there is a one-to-one correspondence of memory bits to display dots.

Fig. 2 shows a simple 64-element display field organized as an 8-word by 8-bit array. For this example, an 8-byte display memory could be used to map the 64-element display. The top left element in the display corresponds to the most significant bit of the first memory byte, and the bottom right element corresponds to the least significant bit of the last memory byte. Fig. 3 shows an example of how the display memory maps to a video character following the format of Fig. 2. Movement left or right is accomplished with shift operations; movement up or down is accomplished with arithmetic operations.

Sync generation - The video controller is required to generate National Television Standard (NTSC) horizontal (line) and vertical (field) synchronization frequencies of 15.750 kHz and 60 Hz, respectively. The NTSC standard is based on an interlaced display. However, since a noninterlaced display is used in this example, these frequencies will deviate slightly from standard in the final system.

Refresh - Display refresh is required because of the dynamic storage characteristic of the CRT. There are two popular techniques for display refresh, DMA and dedicated-hardware controlled. With the DMA technique, the one used in this example, the microprocessor is used as a DMA controller to point in real time at the display memory byte. Since the CDP1802 has built-in DMA control logic, the DMA refresh technique is particularly useful in simplifying the display controller. The main disadvantage is a reduction in processor instruction rate as a result of the "cycle steal" effect of DMA. However, for most graphics games, this reduction has no noticeable affect on performance.

The video-display control functions of video generation, sync generation, and refresh

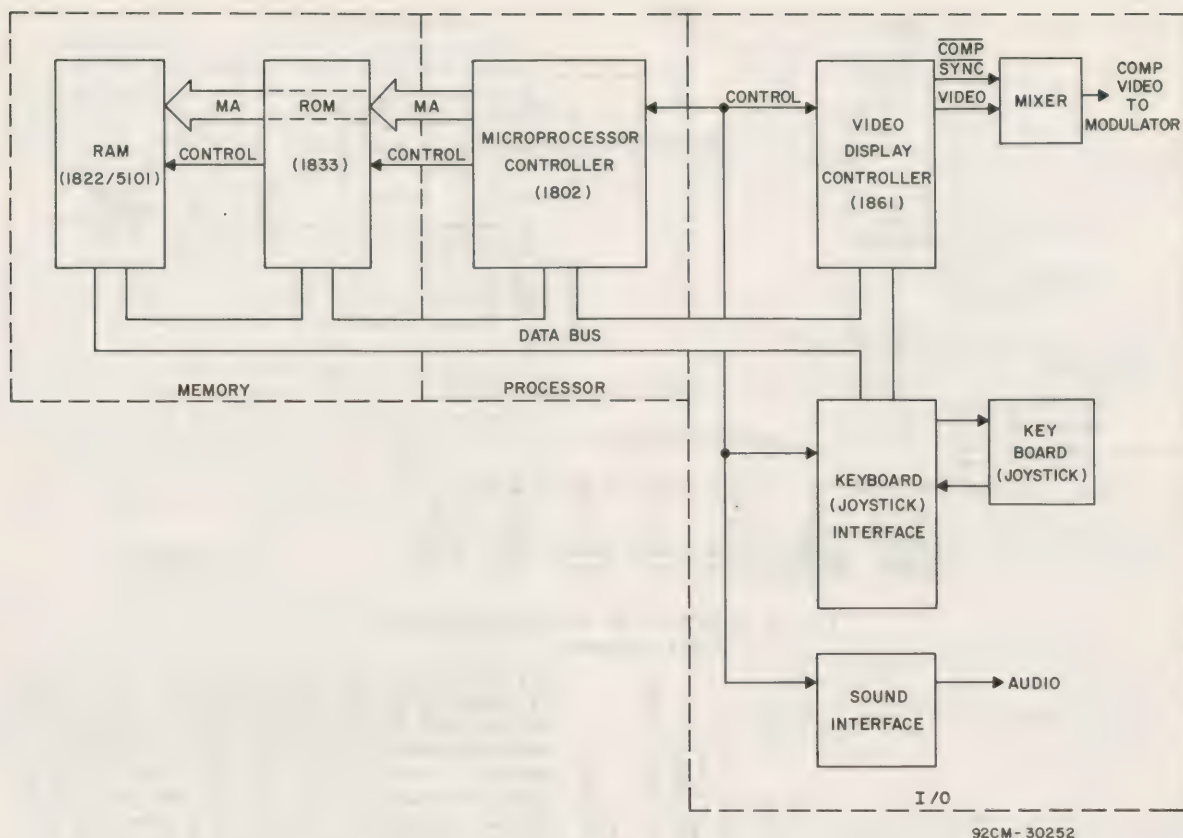


Fig. 1 - Game system block diagram.

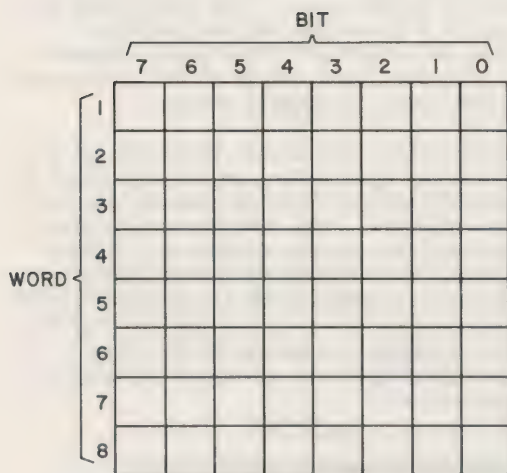


Fig. 2 - 64-element display field organized as an 8-word by 8-bit array.

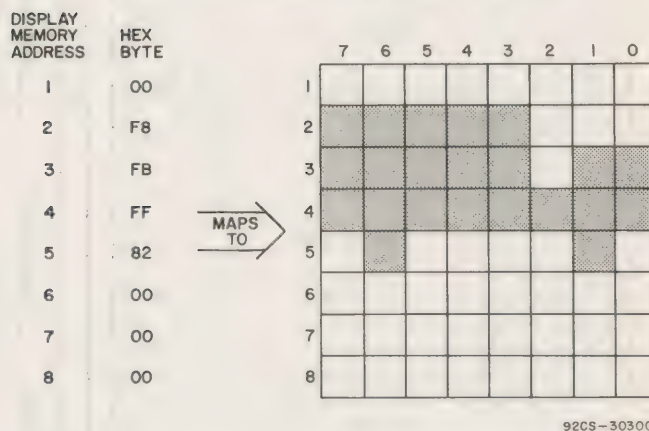


Fig. 3 - Example of bit-mapped display.

are performed by a single CDP1802-compatible I/O circuit, the CDP1861 Video Display Controller. A block diagram of the CDP1861 is shown in Fig. 4.

Fig. 5 shows pictorially the general display format generated by the CDP1861. The display area is delayed 64 lines from vertical sync and has a height equivalent to 128 lines. The bottom of the display area is 54 lines from the leading edge of vertical sync. The

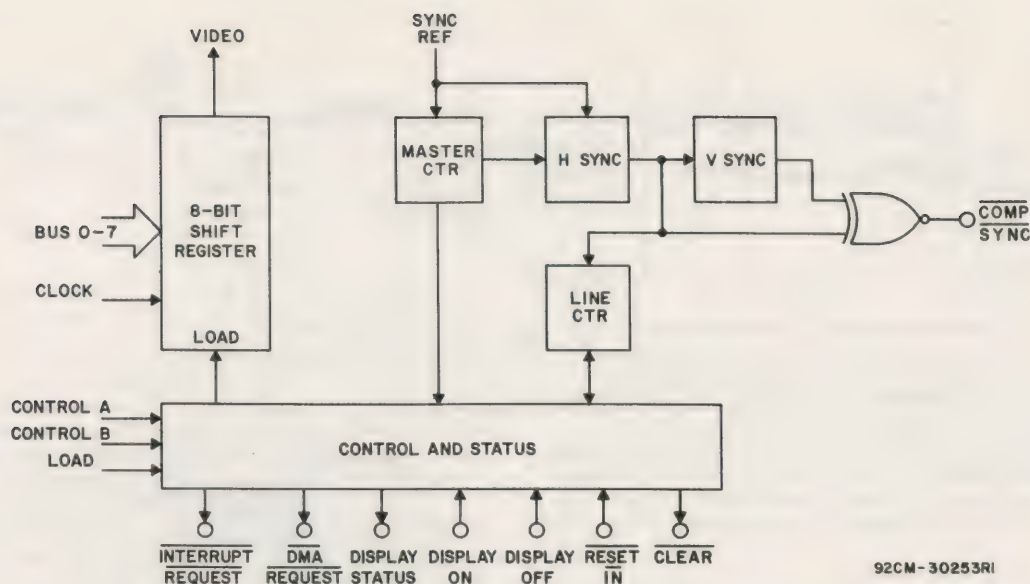


Fig. 4 - CDP1861 TV display controller block diagram.

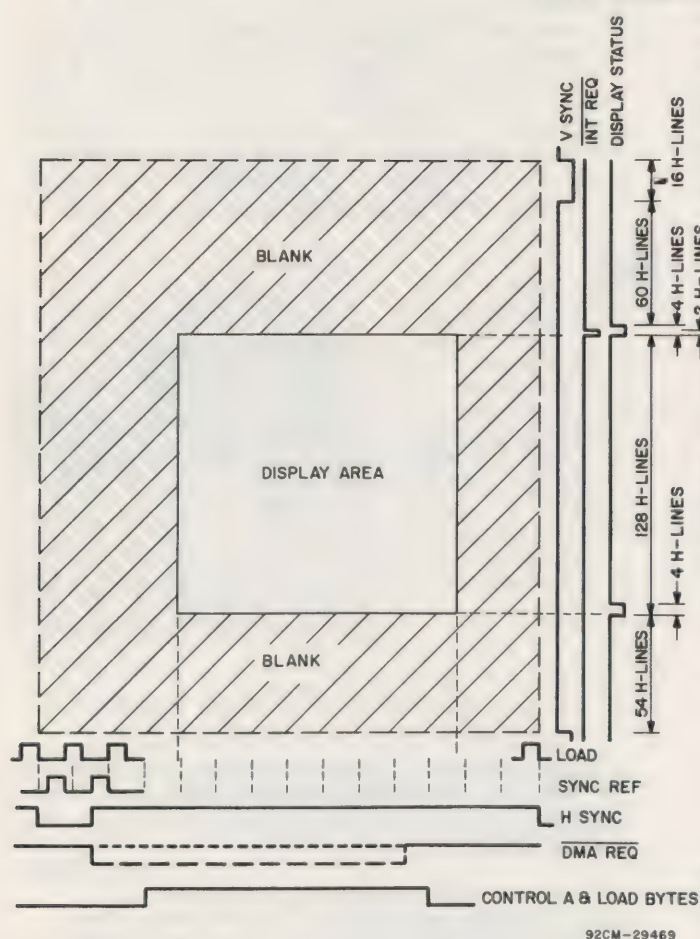


Fig. 5 - Spatial diagram of one video display field (not to scale).

horizontal line is equivalent to 8 bytes or 64 segments in the display area. As a result, the maximum display resolution is 64x128 segments, which corresponds to 1024 bytes of display memory. However, the vertical resolution is under software control of the DMA pointer and can be reduced by factors of 2 from 128 to 1. Typical of many game systems is a vertical resolution of 32 dot rows. This resolution is equivalent to 4 lines per dot and requires that a line be repeated 4 times under software h-control. The resulting matrix of 64x32 elements requires 256 bytes of display memory.

The composite sync signal generated by the CDP1861 from the Sync Reference input creates a 262-line-per-field 60-field-per-second noninterlaced video picture. The noninterlaced picture frame consists of two even fields of 262 horizontal lines. This format differs slightly from the NTSC standard, which has a 525-line interlaced picture frame of one odd and one even field. Table I compares CDP1861 sync frequencies with the NTSC standard for several clock frequencies.

Table I -
CDP1861 Sync Frequencies vs. Clock Frequency

NTSC Standard (Hz)	Clock Frequency (MHz)			
	1.76064	1.764	3.579545/2	
Line	15750	15720	15750	15980
Field	60	60	60.11	60.99

Refresh is controlled by handshake lines between the CDP1802 and CDP1861. The CDP1861 generates an interrupt request (INT REQ) once per field 60 lines after the trailing edge of vertical sync and 2 lines before the beginning of the display window. This request signals the CDP1802 that a display refresh cycle is

Three important system characteristics result from this address interface. First, the direct compatibility of the CDP1833 with the CDP1802 multiplexed address bus. Second, a ROM system of from 1024 to 64k bytes can be configured without any external address-management hardware. Third, the RAM system can be simplified by using the CEO output as a system enable.

A multiple-CDP1833 ROM system is shown in Fig. 9.

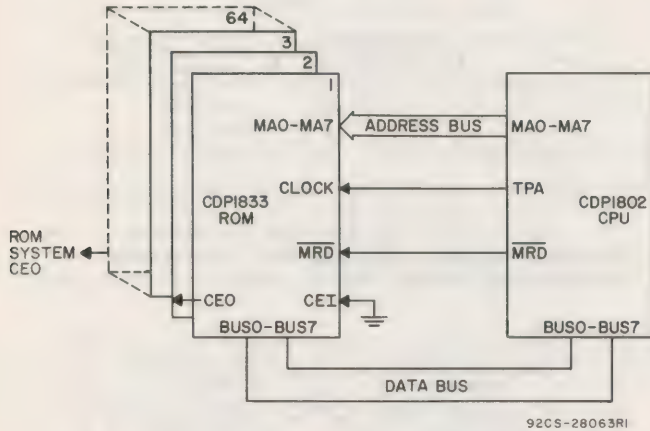


Fig. 9 - Multiple CDP1833 ROM system.

RAM

A 1024-byte RAM system is used for the game application described in this paper. The system, shown in Fig. 10, is designed around the 5101-equivalent CDP1822 256x4 static RAM.

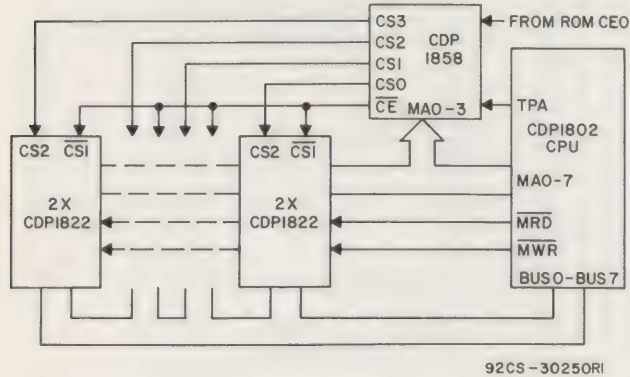


Fig. 10 - 1024-byte RAM system.

As shown in Fig. 10, a CDP1858 is used as an address latch decoder for the system. This device latches the four most significant address bits from the CDP1802 and generates eight decoded outputs compatible with the CDP1822 CS1 and CS2 inputs. The CDP1858 provides enough decoding for a 4k-byte RAM system built with the CDP1822, but only 1k bytes are used in the configuration shown. (System expansion to 4k bytes is easily accommodated by adding CDP1822 RAM chips.)

The RAM system in Fig. 10 does not uniquely decode the sixteen sectors occupied by the 4k-byte RAM. A more indirect but simpler approach has been adopted to enable the RAM system. Recall from the ROM design that each ROM device uniquely decodes its sector address and generates a signal (CEO) indicating selection. By "daisy chaining" the CEI inputs and CEO outputs of the CDP1833's, a ROM-signal is generated. This signal is compatible with the CDP1858 ENABLE input and is used to control the RAM system. It therefore follows that the RAM system responds to any address not occupied by ROM.

Operation and Enhancements

The hardware system is normally controlled by an operating system stored in ROM. As an example, consider the operating system used with the COSMAC VIP to control operation of memory read, memory write, tape read, and tape write functions. Although a tape interface has not been discussed, it is easily implemented via the CDP1802 I/O flag input (EF) and serial output (Q). To use the operating system for memory read, first call the operating system, then enter a starting address (4-digit hex), and finally sequence through memory by pressing any key. The TV display will show the memory address and contents.

To simplify programming of game applications, an interpretive language can be designed. A language called CHIP-8 has been developed for the COSMAC VIP. An instruction listing and description is given in Table II. As a brief example, consider programming the character in Fig. 3 to move from left to right across the display. Fig. 11 shows the program flowchart and code. This example illustrates the simplicity afforded by programming in the game-oriented CHIP-8 language.

The lowest-cost system-implementation method bypasses the operating system and executes game software directly. This method is equivalent to a player-only system.

Enhancements to the basic system include color, programmable tones, and increased resolution. A unique approach to color encoding is available as an add-on circuit to the CDP1861, the CDP1862. This device is capable of creating the illusion of high-resolution color for a 64x32 element display by using a single CDP1822 color RAM to produce eight programmable colors. Programmable tones are made available by means of another I/O circuit, the CDP1863. This device interfaces directly with the CDP1802, generates up to 256 tones in the audio range, and can be gated ON and OFF through a separate input. For future systems, a high-resolution bit-mapped compatible device has been developed capable of 192x128 element resolution (3072 bytes), programmable foreground and background NTSC color, and sync generation. This device represents an evolution of the basic system discussed above and is applicable to an increasing number of sophisticated graphics systems.

Table II — CHIP-8 Instructions

Instruction	Operation
1MMM	Go to 0MMM
8MMM	Go to 0MMM+VO
2MMM	Do subroutine at 0MMM(must end with 00EE)
00EE	Return from subroutine
3XKK	Skip next instruction if VX=KK
4XKK	Skip next instruction if VX ≠ KK
5XY0	Skip next instruction if VX=VY
9XY0	Skip next instruction if VX ≠ VY
EX9E	Skip next instruction if VX=Hex key(LSD)
EXA1	Skip next instruction if VX ≠ Hex key(LSD)
6XKK	Let VX=KK
CXKK	Let VX=Random Byte (KK=Mask)
7XKK	Let VX=VX+KK
8XY0	Let VX=VY
8XY1	Let VX=VX/VY(VF changed)
8XY2	Let VX=VX&VY(VF changed)
8XY4	Let VX=VX+VY(VF=00 if VX+VY ≤ FF. VF=01 if VX+VY > FF)
8XY5	Let VX=VX-VY(VF=00 if VX < VY, VF=01 if VX ≥ VY)
FX07	Let VX=current timer value
FX0A	Let VX=hex key digit(waits for any key pressed)
FX15	Set timer=VX(01=1/60 second)
FX18	Set tone duration=VX(01=1/60 second)
AMMM	Let I=0MMM
FX1E	Let I=I+VX
FX29	Let I=5byte display pattern for LSD of VX
FX33	Let MI=3 decimal digit equivalent of VX(I unchanged)
FX55	Let MI=VO:VX(I=I+X+1)
FX65	Let VO:VX=MI(I=I+X+1)
00E0	Erase display(all 0's)
DXYN	Show n-byte MI pattern at VX-VY coordinates. I unchanged. MI pattern is combined with existing display via EXCLUSIVE-OR function. VF=01 if a 1 in MI pattern matches 1 in existing display.
OMMM	Do machine language subroutine at 0MMM(subroutine must end with D4 byte)

Summary

A programmable game system based on the CDP1802 microprocessor with compatible I/O and memory has been described. The system features a bit-mapped display with DMA controlled refresh and sound. The system is easily controlled by a ROM operating system and programmed for game applications by means of a specially designed interpretive language. The application described is typical of microcomputer designs and illustrates the general design procedure for any specific microprocessor system.

Bibliography

1. User Manual for the CDP1802 COSMAC Microprocessor, RCA Solid State Publication MPM-201.
2. A. Young, "Getting to Know the COSMAC," Electronic Design, October 25, 1976.
3. A. Young, "The CDP1802: A Powerful 8-bit CMOS Microprocessor," RCA Engineer, Feb/March 1977, Vol. 22, No. 5.
4. J.A. Weisbecker, "COSMAC VIP - The RCA Fun Machine," Byte, August 1977.
5. A. Young, "Use of CMOS ROM's CDP1831 and CDP1832 with the RCA Microprocessor Evaluation Kit CDP18S020," RCA Solid State Application Note ICAN-6536.
6. CDP1861 Video Display Controller, RCA Solid State Data Sheet.
7. RCA COSMAC VIP Instruction Manual, RCA Solid State Publication VIP300.

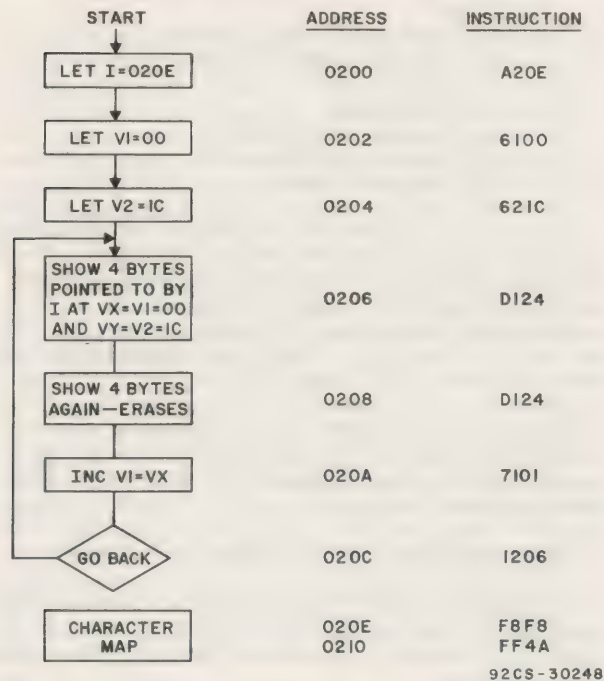


Fig. 11 - Program to move character in Fig. 4 from left to right across display.

A LOOK AT THE FUTURE OF SOFTWARE DEVELOPMENT TOOLS IN THE LIGHT OF CURRENT TRENDS

Richard E. Marsh

Hewlett-Packard Company
Avondale, Pennsylvania 19311

As software projects grow larger and program implementation tools do more work, most of the effort spent on software development will be in the definition and design phases. It is therefore recommended that tools be developed to assist in all phases of development. The software development system is proposed as a unified, extensible tool which will not only assist in all phases of development at the present, but form a foundation for future tools as they become available.

Introduction

The engineering of large software systems has been the focus of much attention in the past few years. Of particular importance are techniques by which reliable software systems can ideally be produced within specification, cost and schedule goals. Unfortunately, there are many obstructions which prevent the achievement of these goals.

Some difficulties which impede software development projects are similar to those problems encountered in other engineering efforts. However, software engineering is a relatively new discipline and many development techniques are not fully evolved. As a result, these problems are experienced more acutely than in other areas:

1. Large projects require large staffs, with their associated communication and management problems. The value of rigid system specification techniques for internal communication is not well understood.
2. Schedule "surprises" are a major problem, arising from inadequate problem definition or blind spots in the design which are carried through until discovery late in the project. To compound the problem, the complexity of a software job is often masked, making it difficult to accurately estimate the time required to complete the task. This masking effect can often be attributed to lack of proper development tools. For example, a relatively minor change may cause a program to exceed memory space limitations, requiring more substantial revisions to be made.
3. Documentation of the design does not always match the actual system characteristics. This can be the result of making many gradual changes to the system which are not immediately reflected in the

documentation, or by relying too heavily on informal (perhaps verbal) specification.

4. Excessive manual clerical effort is required to accomplish simple tasks, allowing many time-wasting errors to creep in. In most areas, engineering time is channeled into producing designs and testing completed prototypes. The software engineer is confronted with the problems of design and testing also, but sometimes ends up handling technical details such as assembly language coding, assembly of programs and filing of listings.

Software Development Standards

The general phases of a software project are:

1. Definition
 - 1.1 Abstract ideas
 - 1.2 Feasibility examination
 - 1.3 User definition
2. Design
 - 2.1 High-level specification
 - 2.2 Detailed specification
3. Implementation
 - 3.1 Program generation
 - 3.2 Program verification
 - 3.3 System release

Each of the phases can be improved by defining and adhering to stringent design standards. These standards should be formalized completely before any actual design is begun, perhaps in parallel with project definition. Changes to standards are not easily accepted if significant amounts of documentation need to be changed to accommodate them.

Experience has shown that, in general, the more standardization, the better. This gives all project members a common language and eliminates some communication problems.

Areas which should be standardized include:

1. Indexing scheme for all phases of project documentation. Assignment of each project area to a node in a hierarchy tree is perhaps the best available technique.
2. High level specification language or diagrams, classified according to the indexing scheme.

3. Detailed specification language or diagrams, also classified according to the indexing scheme.
4. Module or subsystem programmer's guide, by which utility modules for internal use may be specified in a user's guide format.
5. Test specification language or diagrams, again conforming to the indexing scheme.

Of course, it is important that all persons who will have to adhere to these standards understand their reasoning well. Attempts to follow seemingly meaningless procedures prove to be futile, with early breakdown of standards inevitable. Monitor personnel could be assigned to enforce standards, but this could be expensive and not completely effective.

Software Tools

It is considered an accurate statement that anything which saves work without creating more work will gain rapid acceptance (providing that it is economically feasible). This alone is a sufficient rationale for attempting to automatically enforce design standards, as well as alleviating other mentioned development problems.

Any automatic tool for assisting software development is subject to problems of its own. It must be available, reliable and must help, not hinder, the development process. Also, since this type of tool is itself software in nature, it requires computer resources.

These problems are not easily solved. Microprocessor manufacturers provide only very rudimentary support software. Some of this software is less than reliable (perhaps being produced only to make their processor marketable). This restricts the use of advanced automatic tools to those organizations which have enough foresight and resources to produce reliable, extensible tools for their own use.

Conventionally, separate tools are available such that a typical procedure for making a change to a program module is editing, compilation, linking and simulator testing.

Note that the emphasis of the tools is on implementation of the change, with nothing said about its system-wide effects or project level implications. The tasks of updating the detailed system specifications, the high level specifications, the user's manual; assuring compatibility with changes to other modules, notifying other affected project members of the change and perhaps updating the project schedule are left up to the engineer. If one considers a multi-year project, with many engineers and thousands of modules, these details become non-trivial.

A Software Development System

A software development system (SDS) is a unified tool which assists in all phases of the development

process. A simple implementation oriented SDS, called ARCHIVES, has been developed at Hewlett-Packard for internal use. It has had various problems, but serves as a valuable "first pass" by which better tools may be designed.

Basically, the SDS-as proposed is a data base handler which is capable of filing all project development efforts in a hierarchically indexed form. It is also an extensible control framework into which arbitrary tools may be inserted, each of which may access the data base in a controlled manner (Fig. 1). For example, if the tool in question is a compiler, it might be designed to read elements containing source language and create elements containing object code; a source editor may read and write source language elements. Note that access by each tool is not restricted to single elements; given a root element, a tool may optionally reference the entire subtree of that root. In addition, a "current element" may be defined, so that commands like EDIT SOURCE, COMPILE or LINK may implicitly operate on the current element. This prevents errors when a large number of modules are being manipulated, by keeping one's thoughts straight.

The indexing tree structure is defined by a stepwise decomposition of the project into a two-dimensional hierarchy tree [1]. Each node in the tree is assigned a name which describes its function, plus an index number for ease in location

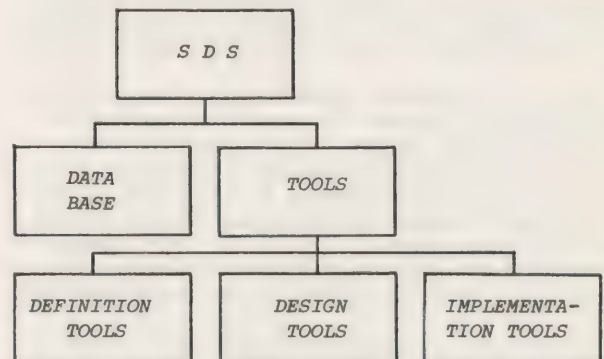


Fig. 1. Software Development System

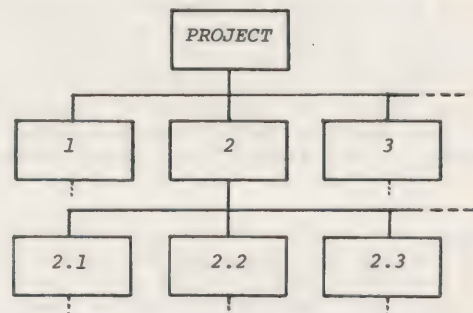


Fig. 2. Decomposition of a Project

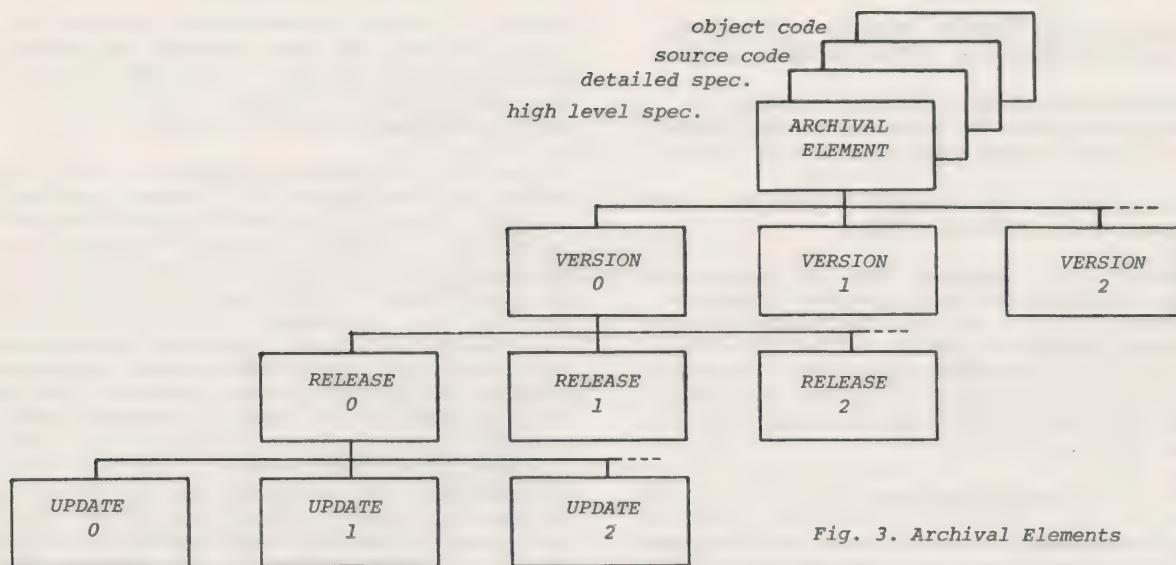


Fig. 3. Archival Elements

(Fig. 2). Since nodes may at first be classified incorrectly, any node and its name must be able to be moved to a different place easily when a more appropriate classification is found. This doesn't cause any problems if all tools make explicit references to the contents of a node by its name, rather than its index number.

Each node in the indexing structure is composed of tree shaped "archival elements", each containing a different type of data, as illustrated in Fig. 3.

Within the archival element, each "version" of that element's contents is stored. A version is assigned a number which is associated with the level of system being described. Initially, all elements are built at version 0. The version number for each element in the system is incremented when the entire system is released. For example, system releases might be made for lab prototype, production prototype, pilot run and production run.

Beneath each version in the tree are "releases," similar in principle to versions. Each release represents a major change within a version. This permits previous releases to be supported while working on a new release.

Beneath each release are "updates," again similar to versions and releases. A new update is produced whenever the contents of a release are changed. Since the number of updates in the system can be very large, and the extent of change between updates very small, a "delta" update approach is recommended to make storage requirements manageable [2].

Although use of this three level description of version, release and update for each module is advised, a two level qualification can be used with adequate results. The ARCHIVES system used a two level approach, but required the system version number to be incorporated as part of the module

name.

Since all of the project documentation is stored in a single data base, security provisions might be desired. These measures would limit access by given personnel to specified parts of the structure. Also, back-up of the data base onto an off-line medium may be expanded to allow moves of seldom used elements to secondary storage, as did the ARCHIVES system.

There is no claim that an SDS as proposed here will solve all of the problems mentioned. Rather, the idea is to supply an extensible framework into which can be fitted almost any tool desired. The point of extensibility must be stressed here, since ever more powerful tools will become practical in the near future. Even in the short term, it would be an advantage to be able to quickly add a different language compiler if the need arose.

Applying the SDS

Each project begins with a number of random thoughts, ideas, and eventually proposals. Some ideas will "fall through the cracks" because their merit cannot be assessed until the project develops further.

The indexing features of the proposed SDS permit each of the idea "modules" to be typed into a data base and immediately assigned to a node. When the project, for example a signal generator, is first conceived, a root node is created bearing the project name, say "IRMA". Into this node is entered a memo stating the need for IRMA in the product line, and perhaps some very general statements about its features.

As the IRMA idea is developed, further nodes are added beneath the root. If feasibility problems are anticipated, an element type "problem" could be appended to each node. At any time, one could in-

quire about the current status of "problem" for a given section of IRMA or of the whole project. Since the level of detail in each node increases as the tree expands, it may be desired to restrict the depth of search if only the general problems are of interest. This is entirely up to the "problem" processing tool which has been provided.

Another valuable element type would be "schedule". Elements of this type could be processed by a PERT/CPM tool [3]. The IRMA root node could contain general schedule estimates, for example, 2 years to production, 2.7 at worst, 1.8 if certain technologies are favorable. As lower nodes are added, schedule estimates for each of these project areas could be used to maintain the overall IRMA schedule.

Eventually, a node called SOFTWARE SYSTEM could be defined. From this point downward in the index tree, nodes would contain archival elements with types such as "detailed specification", "source language" and "object code". A tool could be added to the system so that the user would be prompted for possible changes to specifications and user definition before a change to a source module is made. This would eliminate the time lag between modification of a program and revision of documentation.

Conclusions

Many software tools need to be redesigned from one project to the next. There will be much benefit in considering an extensible support system which will outlast a single project and permit each project which it serves to be developed with better control.

Acknowledgement

The ARCHIVES system at Hewlett-Packard was conceived and implemented by D. R. Hawk.

References

- [1] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Structured Programming. New York: Academic Press, 1972.
- [2] M.J. Rochkind, "The Source Code Control System," IEEE Transactions on Software Engineering, vol. SE-1, no. 4, pp. 364-370, December 1975.
- [3] D. M. Stires and M. M. Murphy, PERT/CPM. Boston: Materials Management Institute, December 1963.

OPTIMIZED DESIGN CYCLE FOR A MICRO-COMPUTER LABORATORY

David M. Robinson

Department of Electrical Engineering
University of Delaware, Newark, Delaware 19711

ABSTRACT

An educational digital system laboratory dedicated to micro-computer based system design is described. The teaching objectives of this laboratory are presented and an implementation which optimizes the system design cycle in this local environment is described.

INTRODUCTION

The continual evolution of digital integrated circuitry to more extensive programmable constructs is largely responsible for an increasing emphasis on digital system design in the electrical-computer engineering curriculum. This design orientation is not a new phenomena; this has been the mode of digital system instruction for the past decade. However, the introduction of micro-processors and programmable LSI support circuits has provided a most appropriate technological environment in which design may be taught as an engineering discipline. The design concepts of requirement analysis, performance specification, system partitioning, establishment of structure and regularity, implementation, evaluation and maintenance are all concepts which are brought sharply into focus under rather ideal circumstances in micro-computer based design.

In this paper, a laboratory environment created to address the issues of micro-computer based design will be described. The discussion will focus on the pedagogical issues as we perceive them, what the laboratory is to accomplish, how the laboratory has been configured and some few examples of current laboratory activities. In order to adequately portray this laboratory function, it is necessary to first describe the background from which students approach their early design endeavors and to define our design cycle.

STUDENT PREPARATION FOR DESIGN

Our digital systems program has been in evolution for over ten years. In its present state, there is a course in this program that intimately relates to the digital system core area available in every term of the student's tenure. Many students take all of these courses while others, electing a less intensive digital systems experience, may take only six or perhaps even as few as the four required digital courses. These courses cover such topics as logic design, switching theory,

computer organization and programming, and the design of computers or computer-like digital systems including micro-computer based systems.

A conscientious attempt is made to teach the digital system core courses from a design point of view. The design instructional philosophy places considerable emphasis on the role of the digital laboratory and all of the core courses are taught with supporting laboratories. These laboratories stress both hardware and software issues. The early laboratories are structured, but the structure is relaxed over time and the third and fourth year labs are executed much in an open laboratory-project oriented environment.

The micro-computer laboratory was developed to strengthen our offerings in the digital system design area. This is a specific two course sequence taken by digital students in the junior year and by the less intensely concerned students in their senior year. Students come to these courses relatively well prepared to address the issues of system design based on micro-computers. As evidence, prior to this course, all of the students have designed hardware at a complexity level of 20-30 SSI chips, are familiar with bus structured systems, have written an interrupt driven mini-computer program, etc. We feel that such a level of maturity is necessary to address the issues of micro-computer design as we perceive them.

The laboratory was originally designed to support the second course of this two term sequence in design. The first course stresses the partition of data and control flow structures of digital systems and the development of alternatives in control structure realization (distributed control, micro-programmed control sequencers, etc.). Hardware descriptions are at the register transfer level and hardware description languages (particularly AHPL) are employed. Interfacing and the design of special purpose interface processors represent the principal case studies. We believe that an inversion of the order of the design course sequence is now appropriate. The micro-computer laboratory should provide a most effective mechanism for the implementation of the notions of design which are introduced.

A WORKING DESIGN DEFINITION

We have implied that there are attri-

butes of micro-computer based system design methodology that are significantly different from other design techniques. What all of these differences in design are and what impact they will all have on the total digital system efforts is a matter for speculation and one which we may just be beginning to understand. Most observed micro-computer based designs have been instances in which the micro-computer was either a replacement for hardwired logic or employed as an inexpensive mini-computer. These are very exciting applications, but we suspect that this may not completely represent the ultimate role which the micro-computer fulfills. In this impression, we have been very much guided by a tutorial on "The Unique Aspects of Micro-computer Applications" given by F. F. Coury [1]. In this tutorial, he offered a definition of a micro-processor which we have modified and extended to act as a guide for development of the micro-computer laboratory. In its new form, the working definition becomes:

Micro-computer based design is the execution of system design employing programmable logic devices which are embedded in an application and share program and data memory, I/O structure, support circuitry, power supplies, and common packaging with the system.

We have come to use this working definition as a guide in our laboratory development. The working definition has provided a way of thinking about the problems presented to the micro-computer designer and a criterion for establishing a design philosophy.

The working definition is not sensitive to technology changes as are definitions which describe the micro-computer as a MOS/LSI processor or as a single chip or a few chip computer or as an inexpensive mini-computer or even as a replacement for random logic [2,3]. The definition is insensitive to temporal changes which are apparent in attempts to define micro-computers in terms of performance or structure (word length, memory space, numbers of general purpose registers, etc.) [3,4]. All of these kinds of definitions must constantly be re-evaluated.

The working definition is insensitive to the properties of the design space. The design space is extensive and expansive. It includes replacements for mechanical devices (cam timers, cash registers, thermostats). It extends through smart typewriters and intelligent test instruments to include multiple arrays of micro-computers in highly complex computing environments. The definition encompasses all of these applications.

Most importantly, the working definition can be used to distinguish micro-computer based design. For example, the implication is clear that design with micro-computers is different from hardware logic design principally because of programmability. The impact on the design process is different at all levels; conception, execution, documentation, support, production and service. Micro-computer design is different from design based on mini-computer structures because of embedment, the intimate level of integration. Of course, designers have been integrating mini-computers in systems for many years. But, mini-computers are fully functional systems. They have their own packaging and power. They may even have pre-wired slots for options which are integrated, software supported and fully documented. In contrast, the micro-computer has a significant impact on design in that it is a construct. Now of course, we may embed a micro-computer in a micro-computer based mini-computer design and integrate that mini-computer in our overall system design. But this casts the micro-computer in a far different role and the integration design cycle is totally different from the micro-computer embedment cycle.

The definition helps us to perceive some major differences in design procedures. When viewed from the perspective of a conventional hardware designer, micro-computer design is a software intensive activity. Of course, mini-computer designers write some applications code, but that is rather casual. Contrast that activity with the micro-computer designer who might write micro-code to implement an instruction set or trade, say, switch debouncing hardware for several instructions in ROM. When viewed from the perception of a conventional systems software designer, the process of designing micro-computer based systems is hardware intensive. Perhaps never before have the imperatives for hardware understanding been placed so much in evidence for these types of individuals. For the high-level-language programmer, the mission of micro-computer based design must seem virtually impossible, at least in terms of presently available languages and support. Their lack of familiarity with real-time constraint on I/O, interrupt structures, bus disciplines, etc. almost precludes their direct entry into this design environment.

The definition allows us to interpret some unique design support functions. The software intensity of the design process requires software development tools. The environment for software development is influenced dramatically by the embedment implied in the definition. Frequently, in the object system, few of the attributes necessary for software development are

available. The system may not even provide terminal support, it is entirely possible that all of its memory is to be in ROM, etc. This implies that tools like cross assemblers, cross compilers and simulators are required support products. As in most new technologies, the new technology itself spawns the development of a number of specialized instruments which support that technology. Many of these instruments come about as a direct result of the embedment of the micro-computer. This has led to the introduction of development systems, ROM and in-circuit emulators, and various probing instruments such as logic analyzers.

LABORATORY-DESIGN CONSTRAINTS

This laboratory is to support instruction in a design area which promises to be very competitive. Successful designs will be most cost effective and successful designs must move very rapidly from concept to production. This state of affairs and the working environment places severe requirements on the system designer and the design laboratory. Design in this area presents an exceedingly sharp set of contrasts between requirements and tools available to assist the designer in meeting those requirements.

One of the major tenets of this design philosophy dictates that designs be software intensive; that is, make software do as much as possible. The designer is challenged to develop software (which is expensive) to replace hardware (which is cheap) and to do this in minimum time for systems which are minimally configured, have no software tools, and are likely to be incapable of supporting any development tools.

Highly optimized designs will be successful designs. The optimization must prevail at even the lowest of levels: packaging, power supply, bus drivers, etc. A clear requirement is to minimize component count by designing for multiple functionality, partial decoding of addresses, and similar "tricks of the trade". The designer is to accomplish this level of optimization while at the same time keeping the design flexible. Design with this level of hardware economy and sophistication must proceed on a system which has none of the comfortable features which help us debug hardware. There is no front panel with switches and lights, we can't put a probe on a register cell since they are buried in a chip, we cannot even stop the processor to examine its state since it is likely a dynamic device.

DESIGN OF THIS LABORATORY

The problems of micro-computer based design execution are the problems of

hardware and software development. In this environment, the hardware-software discipline binding is so very tight that the same designer is likely (in our laboratory always) responsible for both functions. In spite of the inseparability of hardware and software function, the fact remains that there are chips to interconnect and programs to be written and that those two tasks are essentially different in nature and require different support tools. A global consideration in the design of our laboratory is that the support of these two tasks should be addressed as separate issues.

Hardware Support

The laboratory must provide facilities which allow the hardware design philosophy to go forward in as rapid and well supported a fashion as possible. This almost certainly implies a system in which many components are prewired into at least some minimal system configuration. Osborne [5], in describing the micro-computer design cycle, suggests that as soon as micro-computer selection has been made, the design effort should focus on a development system. He further states that, "At the center of any hardware development system, there will be a box that looks like a mini-computer". We must confess that our system looks like a mini-computer! Indeed, it can be expanded into a rather substantial mini-computer. It is packaged; it has a power supply which is an overkill for a minimal system and can supply a fully configured system containing up to sixteen printed circuit cards inserted into pre-bussed slots. It has committed I/O structures; it is capable of supporting A/D-D/A converter cards. It does not have a front panel with switches and lights, but, even in its most minimal configuration (\$433 + lots of student labor), it does have the ROM capability to support front panel emulation using a terminal (\$1500). It is quite a general purpose device; in short, it is a total abomination of virtually every good micro-computer based design issue discussed in the early parts of this paper. It is our "development system".

The minimum system configuration consists of the "box" with a single card. The chip set chosen for this development was the Motorola M6800 family. The single card system has an MPU, support circuits (clock, reset circuitry, etc.), 1024 bytes of ROM, 512 bytes of RAM, an ACIA (serial communication device) and a PIA (a parallel I/O chip). If system expansion is required, the MPU bus structure is buffered and repeated through the remainder of the system using a second card. This second card also has ROM (EPROM-2708) and an additional ACIA whose function will be described under software support.

The form of this system is dictated principally by economic constraints. We understand the arguments that suggest that the pedagogy might be better served by presenting the students with a chip set and patching board [8]. We would counter, that with those techniques, even for simple experiments, the students spend most of their valuable laboratory time in the meaningless routine of establishing minimal configuration interconnections - they very seldom get such systems working. These laboratories must provide meaningful experiences requiring indepth involvement with complex systems over extended periods of time. Laboratory budgets quickly get out of hand if one attempts to establish that kind of design environment by distributing chips. The student lab station (development system) allows us to "time-share" a substantial portion of generally supportive hardware over our volume of students at minimum cost. Provisions are made via ribbon cable connectors to quickly connect this shared hardware to hardware extensions which the students may specifically tailor to their problem and which may economically be retained over an extended time period.

The ribbon cable connections may make the buffered bus signals available or may be connected to the I/O registers of the PIA. Thus, simple, workable systems can be configured with a single card while more extensive systems may be connected to the system bus which is identical to the conventionally buffered MPU bus. All of the addresses of the two card system are in the upper half of the available address space. Hence, as long as systems require no more than 32K bytes, they can be configured completely devoid of any interaction with the built-in hardware support functions, if that is desirable.

This system is hardly what designers of third generation hardware development aids would call a development system. Moon and Gray [6] outlined the features which a development system should exhibit in order to be transparent in the design process. For our design environment and for the class of systems which must be designed, we believe that this system meets all of the outlined criterion; namely, the system can run in real time; all system hardware during development is the actual system hardware in its final form; the development system only restricts system architecture in the sense of constraining address space to 32K bytes (not a severe restriction in this environment); the development aid in no way restricts program use of architectural features unless the designer wishes to take advantage of some built-in features; the aid can take advantage of interface hardware and resident software but the system design can proceed without requiring those features; and finally, abnor-

mal loading and speed constraints are not imposed by the fixed parts of the system.

Some of the special sub-systems developed using this scheme may be committed to wire-wrap prototype modules and inserted into the system back plane. (We have used standard DEC connectors and module configurations which give us some economic advantage for wire-wrapable cards, card extenders, etc.) As these modules prove to be of general utility, their design may be committed to printed circuit cards and added to a growing family of system level components. Items of this class include: 4K memory modules, A/D-D/A converter cards, multiple terminal serial communications cards, etc.

It should be noted that no mention has been made of interfaces which drive the usual development system peripherals. Broderick [7] advances the cause that software development, at least code preparation, translation, and verification, progresses on the development system. This places severe constraints on the hardware availability of the development system. For example, if we were to consider even adding minimal software features to our student lab station (say a simple text editor and resident assembler) we would dramatically impact the transparency of the development hardware. The current trend in development systems seems to be to build two processes into the same box - one to control the software development portion of the cycle and to interact as needed during the de-bug phase - the other to act as the processor in the object system. We believe that the objectives of such a system are met fairly well in our student laboratory system by the partitioning of memory into development space and user space. The user is free to take advantage of any of the development system attributes which are to be incorporated in the final design or omit such features entirely.

Software Development

The range of tools and techniques and the range of associated costs for software development are dramatic. There are indeed systems which are brought to fruition by the process of hand coding with no tools; while this procedure has low capital costs, the programmer intensive costs might be very high. The procedure does work and it has many advocates as evidenced by a conference on Bench Programming Methods scheduled in Philadelphia in June of 1978. In an effort to reduce the severe demands on programmer development time, many vendors provide development systems with disk and terminal support for a capital investment of about \$20K. Perhaps at the extreme end of the spectrum are the tools developed or currently under development by the Na-

tional Software Works [9]. This is a distributed processing facility implemented on the ARPA network in an effort to discourage the proliferation of redundantly implemented software development tools by providing designers uniform access to an extremely large set of tools distributed over many dissimilar host systems.

In this laboratory system, the link between hardware and software development is provided by that nearly forgotten ACIA (serial interface) in the hardware development system. Software development proceeds on totally separate computing facilities. This possibility has many potential advantages. Obviously, a separate facility may be more appropriately configured to accomplish software development. Perhaps the economics will allow this facility to be larger (more memory) and better equipped with resources (larger and faster disk) especially if it is a shared resource among several design groups. How much larger and better supported with peripherals is a subject for debate or research but there are multiple user time-sharing systems running on micro-computer based systems which could certainly do the required task. At this institution, we have used three different systems (two different operating systems) to accomplish these tasks. Software support tools have been developed using DEC's BASIC+ running under RSTS on both an 11/50 and 11/70. These have both been satisfactory mechanisms for developing adequate software support. Software tools have also been developed on an 11/70 running the UNIX (Bell Laboratories) operating system. This later alternative provides a somewhat richer environment and is more the current mode of operation and the one which will be described.

Conceptually, the software development system is quite simple. The student lab system is configured with an ACIA which permits that station to communicate with a central time-shared facility. The hardware of our implementation permits this communication to be hardwired or coupled over telephone lines. A program in ROM (actually the EPROM of the second system card) supports a talk-through mode of communication so that the laboratory system terminal can communicate through the lab system to the central facility. A user can thus take advantage of all the large system features of the central facility (good text editors, file systems, cross assemblers, cross compilers, etc.). A more complete discussion of available tools is included in [10]. An absolute load module is produced on the central facility and down line loaded into the student's station; the ROM program transfers control locally to the now console terminal.

The users perception of this system

is that the minimal micro-computer is supported by a substantial disk-based operating system and the only real programming limitation is the available memory space for executable code. The user can obtain listings with cross-references, accessibility to a potentially large library of subroutines and all of those pleasant aids to software development which are very expensive to provide in a resident development system.

It may be that clearly separating the tasks of hardware and software development in this situation has led to a system configuration which is richer in their joint development than attains when the two tasks are forced together in a development system. Generality is not necessarily implied; this may be only a situation which is optimum for our peculiar local environment.

THE DESIGN LABORATORY

As previously indicated, laboratories at this level are rather unstructured. There are scheduled lab sessions when consultation with graduate assistants and student group discussions are encouraged but the general mode of an open laboratory environment is established for this activity. There are, however, two required experiments which begin the micro-computer lab experience. The first of these is designed to establish the student's relationship with the laboratory. The student is required to go through the regimen of logging into the UNIX system, creating and editing an assembly code file, assembling that file, and loading, executing, and interactively de-bugging the code. The particular problem is not specified except that results should produce some generally useful program or subroutine which can be added to our public library; that criterion is set to establish the level of documentation required.

The second compulsory experiment stresses, perhaps artificially, the replacement of hardware with software. The student has four characters of seven segment LED display and an undecoded matrix switch. The student can use integrated circuits for buffering to establish electrical signal compatibility between the PIA of the single card system and the external devices, but no additional hardware is allowed. The object of the experiment is to display the last four hexadecimal characters entered from the keyboard. The student must write code to multiplex and refresh the display, look for new character interrupt inputs, and debounce key contacts within the constraint that they have only the two eight bit I/O ports provided by the single PIA device. As background, they are directed to a simpler but similar switch

decoding problem in the M6800 Applications Manual [11] and a related display section of that manual. Many of the students find this a rather intensive learning experience.

With the "compulsory" experiments accomplished, the students enter a "free-form" or project oriented phase of the laboratory. The mode of operation switches from individual to small group efforts. Here, the students are encouraged to suggest projects of their own and if they seem reasonable and do-able, they are encouraged to go to completion. We seem to be continually operating in a backlog mode with more projects suggested than can be accomplished.

There are two general classes of continuing projects which we refer to as micro or mini projects depending on their complexity. A micro project can be all hardware or all software depending on the students interest but a mini project should generally demonstrate broader competences. Typical micro projects are: design a complex clock, extend the keyboard - LED display to a calculator, develop control software for a PIA driven D/A and comparator to implement an A/D converter, use a single input line and internal timing to support serial communication, design arbitrary analog function generators, or use the A/D-D/A sub-system as a memory scope.

Many of the mini projects have been directed at hardware and software supported enhancements of the micro-computer system itself. These include the development of a 4K RAM module for the system, a module for ROM programming, the A/D-D/A converter module, a module which provides multiple ACIA facilities for communication, a module which provides address trace detection, a programmable interval timer, automatic and remote reset functions, and multiple level vectored interrupts. Of these, only the first four have proved to be in sufficient demand to have been committed to printed circuit cards; the remaining circuits are executed at the wire-wrap module stage.

As often happens, systems developed for one particular operation contribute to efforts in other areas. We have had an ongoing interest in computer networks and the development of this system has had an impact on those efforts. A communication concentrator is being designed about this system as is a protocol observer (rather like a logic analyzer for serial data). Several systems are being fabricated for other departments on our campus for use principally as inexpensive mini-computers in laboratory data acquisition and control environments.

CONCLUSIONS

The partitioning of the hardware - software development tasks and the methods of supporting the micro-computer based system design cycle have resulted in an optimized design procedure for this university environment. We believe that the recognition of significantly different micro-computer development tasks and the independent optimization of tools to respond to those tasks has allowed the economic development of a laboratory which can support a large number of students in an intensive micro-computer based design effort.

ACKNOWLEDGEMENTS

Many hours of student effort have gone into the development of this laboratory. In addition to classes in EE323 and EE667, I would particularly like to acknowledge the contributions of Bob Huckins (many of his original ideas in [12] formed the basis of this lab), Dave Sincoskie, Rich Hammond, Gary DeAngelis, Ken Bunting, Art Stadlin, Steve Folsom, John Kallal, Mat St. Peter, and Tom Levis.

REFERENCES

- [1] Coury, F.F., Unique Aspects of Micro-computer Applications, Compcon 76 Tutorial, IEEE 76, CH1070-2.
- [2] Williman, A.D., & Jelinek, H.J., "Introduction to LSI Micro-processor Development", IEEE Computer, June 1976.
- [3] Lalot, T.A., "Micro - processors Present and Future", IEEE Computer, July 1974.
- [4] Agerwal, T., Masson, G., Westgate, R., Designing with Micro-processors, Compcon 76 Tutorial, IEEE 76, CH1178-3.
- [5] Osborne, A., An Introduction to Micro-computers, Vol. II, Osborne & Associates, Berkeley, Ca., 1976.
- [6] Moon, J.B. & Gray, L.C., "Hardware Aids: Software Overtones", Compcon 75.
- [7] Broderick, W.J., "MDS: An Advanced Development Tool", Compcon 75.
- [8] Wakerly, J.F., Logic Design Projects Using Standard Integrated Circuits, John Wiley, N.Y., 1976.
- [9] Fleischman, S.L., "Software Development for Mini-computers Using National Software Works Tools", Compcon 77.
- [10] Hammond, R., Sincoskie, D., Minnich, R., "A Software Development System", Proceedings Micro-DELCON, Mar. 1978.
- [11] M6800 Micro-processor Applications Manual, Motorola Semiconductor Products, Inc., Phoenix, Az., 1975.
- [12] Huckins, R.J., "a Laboratory Micro-processor System", Master Thesis, Univ. Delaware, June 1977.

A SOFTWARE DEVELOPMENT SYSTEM

R. Hammond, W. Sincoskie, and R. Minnich

Department of Electrical Engineering
University of Delaware, Newark, Delaware 19711

ABSTRACT

The general attributes of effective microcomputer software development systems are discussed. The need for high level language cross-compilers is presented. A description of the microcomputer software development system at the University of Delaware is given and three current projects to implement cross-compilers are discussed. Performance comparisons with another currently available microcomputer software development system are given. Conclusions about the suitability of the systems compared for software development and recommendations on the form of future software development systems are made.

I INTRODUCTION

While microcomputer hardware is well into its third generation, microcomputer software development is still in its infancy. Software development costs make up an ever increasing part of the total microcomputer system cost, and are engendering numerous proposals to deal with the situation. While considering these proposals, we realized the need to form an overview of microcomputer software development. Since it appears that microcomputer software is suffering the same problems which beset minicomputer software development, we may be able to profit from the minicomputer experience. One lesson to be learned from solutions to minicomputer software problems is the distinction between software development systems and end product systems [1-3].

This paper develops the end use versus development distinction for microcomputer systems and considers the implications of this distinction. The following section develops both the hardware and software characteristics of a software development system. Section III discusses higher level languages and their place in a software development system. In section IV, the software development system in use by the Department of Electrical Engineering at the University of Delaware is described with reference to the characteristics given in the second section. Section V describes three current efforts to implement higher level languages for microcomputers in the Electrical Engineering Department. The performance of the department's software development system is compared with a commercial system in section VI. The last section contains conclusions and recommendations for future areas of effort in

software development systems.

II DEVELOPMENT SYSTEMS

A software development system is a tool to facilitate the production, maintenance, and documentation of target machine software. A software development system which is natural to use should be consistent, simple, responsive, and complete. A system which is consistent uses the same method of doing a job in all situations. In general, consistency is achieved by separation of functions into basic routines which may then be used together in whatever combination is needed.

Simplicity implies the system's organization is regular and the method of invoking utility programs is concise and clear. If the invocation requires a specification of the state of all possible options, it is probably a complex procedure. If the invocation assumes a reasonable default state for the options, it is generally simple. Simplicity complements consistency by making it easy to use the separate routines together to do complex tasks.

A responsive system is an interactive system which completes tasks in a reasonable amount of time. Obviously, reasonableness is based both on the user's patience and the task being performed, generally it seems that less than sixty seconds to complete a task is reasonable. This figure is based on personal observation and seems to be about the amount of time a person will wait before going for a cup of coffee or starting some other activity.

A complete system is one which has all the software tools needed to maintain itself. A good list of software tool requirements for a software development system is provided by Fleischman [3]. The most important requirements are that provisions for assemblers and compilers, debugging aids, text editors, and document preparation facilities must be made. Fleischman also points out that the software development system is completely independent of the target machine. The development system may run on the target machine, a similar machine, or even a completely different machine.

The requirements given above for a good software development system both pose a problem and suggest a solution to that problem. The problem is that the high

speed peripherals needed to support the development system's software are not usually required or even available in the end use system. The solution is to separate the hardware necessary to support the software development system from the target machine development hardware. This solution permits the software development system to become a multi-user system which both distributes the cost and provides better utilization of the expensive high speed peripherals.

Another advantage of separating the development system from the target machine development hardware is the capability of supporting several target machines with one software development system. After all, most of the development tools are independent of the target machine, since only the code generation part of compilers and assemblers depend on the target machine architecture. Thus the entire cost of a new development system is not incurred for each new microcomputer. The effort involved in relearning a new operating procedure for a new development system is also avoided.

The separation of software development from target machine hardware means that the target machine hardware can be configured as needed in the end use system. The only extra requirement for the target machine hardware is a link with the software development system to enable the loading and execution of programs in the target machine's memory. The link with the target machine could be as simple as an asynchronous serial line with a program in ROM to read in the object code and store it in the proper memory area. Such a capability is provided by most manufacturers' console emulator ROM which also provides the ability to start a program which has been loaded into memory. At the other extreme, the link could be as complicated as an in-circuit-emulator with a controlling processor which communicates with the development system.

III HIGH LEVEL LANGUAGES

Time is money is an old cliché, but it is very appropriate in the new area of microcomputer based products. Not only is the time to design and produce the product important, but so are the times to modify and repair it. Programming in higher level languages with the aid of good software development systems can produce products with lower design, maintenance, and modification costs [3-7]. Programming in higher level languages on a substantial software development system provides an optimum environment and is the mode of operation discussed below.

The time to develop a microcomputer program may be broken down into three basic

areas: design, coding, and debugging. Design is the statement of an algorithm to perform the required function. Higher level languages encourage the designer to use top-down design by providing the automatic context control and control flow structuring needed to make top-down design easy. Control flow structuring permits top-down design by allowing the designer to break the problem into what Lewis and Saunders describe as "mind-sized tasks" which may be expressed as subroutines or procedures [4]. Automatic context control removes the designer's need to keep track of such details as the calling requirements of subroutines and the contents of the processor registers. This permits the designer to spend his time on the structure of the algorithm and not its implementation details.

Coding is the translation of the design into a machine understandable language. The translation is assisted by the ability of high level languages to express constructs which are often used in designs, such as while-do, if-then-else, and repeat-until. These constructs are automatically handled by a compiler removing the errors and wasted time associated with manual translation into assembly language. Further, the close match between the high level language constructs and the design make documentation easier. The control flow is obvious from the program and comments only need to explain why the particular structure is used.

High level languages also assist coding through their data structure declaration facilities. The advantage of explicit data structure declarations and obvious data references are the information they convey to a reader of the program. Generally, $A[I] = X$ is easier to understand than the indexing polynomial evaluation required to get the address of $A[I]$. While the assembly language programmer may explain how he uses the data space, it is not required to get his program running. On the other hand, structured high level languages force the programmer to declare his variables and their use before he can get his program to compile. Thus, high level languages provide more complete documentation in the program itself (where it isn't easily lost).

Debugging is the modification of the program to produce the desired result. This phase of program development is the one most influenced by the software development system. Efficient debugging requires such editing capabilities as moving blocks of the program to other locations, insertion or deletion of text, and global modifications of text form. Debugging also makes full use of high level languages' context control. It is much

easier to change a subroutine when the change will not affect any calling routines by changing their context. A primary need in debugging is for rapid turnaround time on program changes, since there are likely to be a large number of small changes before the program runs properly.

At this point a warning is required. Although high level languages provide advantages over assembly language programming, they are not yet well implemented for many microcomputers. The cost of providing high level languages will remain high until the availability and portability of compilers is improved and the effort required to modify a compiler for a new target machine is reduced. Further, the size of code generated by the currently available compilers is higher than hand coding in assembly language. Happily, if history is any guide, optimizing compilers will become available in the near future. At the present, a partial solution might be to code the awkward or time-consuming parts of the program in assembly language, leaving the major portion of the code in high level language. This approach has been explored by both Holthouse and Cohen and Lewis and Saunders [4,5].

IV THE UNIVERSITY OF DELAWARE'S SYSTEM

The software development system reported here has been implemented with the aid of the UNIX operating system developed at Bell Laboratories by Ritchie and Thompson [8]. The UNIX system at the university runs on a PDP 11/70 configured with a rich assortment of peripherals, many of which would not be needed in a system used only for development. The basic UNIX system provides several of the requirements of the ideal system outlined above. A text editor named 'ed' is the standard editor distributed with UNIX. 'Ed' is extremely powerful in terms of what can be done with a single command, while the commands necessary to enter a document or program and make simple changes can be learned in less than an hour. Documentation aids in the form of 'roff', 'nroff', and 'troff' are available and can typeset documents for output devices ranging from a lineprinter to a phototypesetter. We have actually followed our own advice and prepared this paper (which qualifies as microcomputer documentation) using 'ed' and 'nroff'.

Unfortunately, the authors of UNIX did not see fit to provide niceties like cross-assemblers and cross-compilers for the microprocessors in use at the university, and so some of the work was left to us. In particular, basic tools such as a cross-assembler, cross-compiler, link-editor, and a relocater are needed. The ordering of the tools in the previous sentence is significant, since it is the in-

tended order of implementation.

A cross-assembler for the M6800 processor was written about a year ago, and is the basic programming tool currently in use. A simulator was implemented at about the same time, and is also being used. The high performance of the cross-assembler, and the relatively small size of programs being developed for microprocessors are the main reasons a link editor and relocater are low priority items.

This UNIX system constitutes the host machine for our software development system and is the major part of the system hardware. The target machine used at the university is an internally designed and constructed microcomputer system based on the Motorola M6800. Each basic system consists of a CPU card, prewired space for up to 15 additional cards, a power supply, and a mounting box. The CPU card supplies a limited stand alone system consisting of a M6800 processor, 1024 bytes of ROM, 512 bytes of RAM, an ACIA (serial communications line) with RS232C and TTY interfaces, 16 parallel I/O lines, and assorted support circuits. Included in all of the systems are two additional cards, a bus buffer card and a 4-K RAM card. The bus buffer card functions as a buffer between the MOS compatible signals on the CPU card and the tri-state bus used by the rest of the cards. In addition, it provides a second ACIA and an EPROM to replace the ROM on the CPU card. The 4-k RAM card contains 4096 bytes of static RAM.

There are two modes of operating the software development system. The first requires two serial ports on our development computer. The development computer runs a program which provides a software connection between two ports. With this, the user at a terminal connected to the development system is attached to the console port of the microcomputer. To load a program into the microcomputer involves giving it a load command, then causing the development system to dump a properly formatted load module into the microcomputer. The load module was, of course, created earlier by the cross-assembler or cross-compiler. The program in the microcomputer may then be run and debugged using the console emulator ROM in the microcomputer.

The second mode uses both ACIAs on the microcomputer. The one on the CPU card is connected directly to the console terminal, while the bus buffer card ACIA is connected to a serial port on the development system. To communicate with the development system, the user runs a talk-through program which resides in the console emulator EPROM. To load a file a program is run on the development system which sends an escape sequence followed by the load module

to the ACIA on the bus buffer card.

V. CURRENT EFFORTS

Currently, there are three projects concerned with bringing up cross-compilers under UNIX. The first, and conceptually easiest idea was to bring up one of several commercially available cross-compilers. Many of these products implement one of several PL/1 derivatives and are usually written in FORTRAN to ease transportation problems. Unfortunately, FORTRAN is not really transportable. Problems arise dealing with 'extensions' to FORTRAN, and also with the size of the cross-compiler. The PDP11 has a maximum address space of 65536 bytes, while most cross-compilers written in FORTRAN are several times this size. Thus, the cross-compiler must be either broken into passes, or overlaid, neither of which is a trivial task. Problems also arise with integer constants which cannot be represented in 16 bit form. The PDP11 and most similar systems have a word length of 16 bits, which is the cause of both the address space and integer size problems.

One pre-release version of a cross-compiler was worked on for a significant amount of time and was finally abandoned. We were unable to compile this product as received using several PDP11 FORTRAN compilers, Burroughs B6700 FORTRAN, or even IBM's OS/VS2 FORTRAN. Non-standard 'extensions' used in the cross-compiler were the cause of most of the problems. After much effort, and significant modifications to the cross-compiler, the UNIX FORTRAN compiler, and even the UNIX operating system, the cross-compiler finally compiled, producing an object module that was too large for the PDP11 address space! Since the cross-compiler code was not set up in a manner that would make it easy to overlay, or to break it up into several passes, the effort was abandoned.

We have been much more successful with a commercially available cross-compiler purchased from WINTEK [11]. This product was overlaid, all constants were expressible in a 16 bit word length, and it used ANSI FORTRAN. Even a product of this quality required several modifications and lengthy debugging sessions of UNIX Fortran before it would compile, and an overlay loader had to be written for it to execute. It is currently being tested.

The other two projects represent a very different approach to implementing a cross-compiler. The system programming language supported by UNIX is named C [9]. The C compiler source is released with the UNIX system and rather than write an entire compiler, we decided that altering the C compiler would allow a more rapid introduction of the language.

At this point, a few words about the structure of the C compiler are in order. The C compiler is written in C as two passes, with an optional third pass. The first pass is a syntax and semantic analyzer. It constructs expression trees which are then output in reverse-Polish notation to an intermediate file. This notation, along with some additional operators produced by the first pass, is referred to as the intermediate language. The second pass reconstructs expression trees from the intermediate language which are isomorphic to the trees constructed in the first pass. These expression trees are then operated on by an expression optimizer and a code generator. The second pass outputs assembler code which can be optimized by the optional third pass. The resulting code is then assembled to produce the final machine code [10].

The compiler can thus be broken into several fairly independent pieces, which can be identified as dependent on source-language, intermediate-language, or target machine language. If the intermediate language is left unchanged, a new source language can be implemented or a new target machine language added by altering only a fraction of the compiler. The resulting multi-compiler is transportable (since it is written in one of its languages) and can compile any of the source languages for any of the target machines while running on any of the target machines.

We have actually done this in two cases. In one case, a PASCAL syntax and semantic analyzer was written and in the other, a code generator for the M6800 microprocessor was written [12]. The three resulting PDP11 based compilers (C for M6800, PASCAL for PDP11, and PASCAL for M6800) are now being tested. At present there are no plans to transport or test the four M6800 based compilers (C to PDP11, C to M6800, PASCAL to PDP11, and PASCAL to M6800), although it would be interesting to cross-compile for the development machine on the target machine.

VI. PERFORMANCE COMPARISONS

This section is intended to give an informal price and performance comparison between a software development system as described above, and a commercial microcomputer software development system. The commercial system was picked on the basis of being one of the most popular systems in use today, and also because it offers one of the few available resident compilers. We would like to emphasize that this comparison is extremely informal, and that it is beyond the intent and scope of this paper to discuss performance measurement of computer systems.

Table I contains some performance measurements for a few typical software development utilities. The commercial system on which the sample utilities were run is configured with the maximum memory and fastest diskettes available. The software development system measurements were made on the department's PDP 11/70 system, running with one user. Measurements of the 11/70 indicate that there is a linear relationship between actual execution time and simultaneous job load. This means that if five compilations are executed simultaneously, each execution takes five times longer to finish than when run as the only program. It may be noted that the percentage of time spent compiling as compared to editing and debugging is very small, and that even on a 16 user system, one rarely experiences more than two compilations running simultaneously. The figures in Table I are given as raw data, but a more meaningful comparison can be extracted by comparing the real-times to perform a specific job (such as a development iteration: edit, compile, and execute). In Table I, the commercial system performed all jobs for the 8080 target machine, while the 6800 jobs were run on the PDP11/70.

TABLE I

Task	Target Machine	Source Size (stmts)	Object Size (bytes)	Time List	Time Nolist
Copy	8080	13900 bytes	-	-	:20
Copy	6800	13900 bytes	-	-	:02
Assemble	8080	313	600	4:25	1:50
Assemble	6800	307	571	:05	-
Compile	8080	2	0	1:19	1:10
Compile	8080	227	1100	4:40	3:00
Compile	8080	1250	3988	12:55	8:15
Compile	6800	3	0	-	:13
Compile	6800	359	7350	-	:24
Compile	6800	898	15570	-	:39

VII CONCLUSIONS

Perhaps fortunately, our experience did not follow the order presented in this paper. We needed a development system to support the course in microcomputers and because of time and money constraints utilized the already available UNIX system. We then found by experience with other systems how well ours worked. Thus, we are attempting to explain why our system works, and justify why the same approach may be successful for others. We have tried to present similar arguments from other authors for minicomputer software development because we expect that we can learn from

this repetition of history.

The raw data in Table I may be manipulated to support either single-user stand-alone systems or multi-user systems like UNIX. However, having experienced both systems, we are very pleased with our systems performance. We believe that a multi-user software development system is cost effective in situations with several concurrent projects and that future software development systems will have a similar form.

A major problem with current software development systems is the high cost of developing them. We have pointed out several approaches which would help reduce the cost of a new development system. The separation of the development system from the target machine provides the capability to support many different machines without a totally new system for each. The use of high level languages allows the transportation of useful development system tools to new systems. Compilers using the same intermediate language permit the same source language to be translated for many target machines without writing a complete compiler for each. All these approaches depend on planning ahead enough to make it easy to reuse the current system as much as possible, to avoid writing duplicate material.

REFERENCES

- [1] D. L. Mills, "Executive Systems and Software Development for Minicomputers," Proceedings of the IEEE, Vol. 16, No. 11, pp. 1556-1562 Nov. 1973.
- [2] H. E. Pike Jr., "Software Production for Minicomputers," Proceedings of the IEEE, Vol. 16, No. 11, pp. 1544-1556, Nov. 1973.
- [3] S. L. Fleischman, "Software Development for Minicomputers Using National Software Works Tools," Compcon 77 - Fall.
- [4] L. E. Lewis and J. L. Saunders, "Using High Level Languages to Produce Load Modules for ROM in Medium (2000) Quantities," Compcon 77 - Fall.
- [5] M. A. Holthouse and J. B. Cohen, "High-level Language Programming for Mini- and Micro-Computer Systems" Microcomputer '77.
- [6] Y. P. Chien and H. M. Driezen "A High-level Language for Microprocessor Applications," Microcomputer '77.
- [7] M. D. Maples and E. R. Fisher, "Real-Time Microcomputer Applications Using LLL Basic," Computer, Vol. 10, No. 9, pp 14-21, Sept. 1977.
- [8] D. M. Ritchie, K. Thompson, "The Unix Time-Sharing System," CACM, Vol. 17, No. 7, July 1974.
- [9] D. M. Ritchie, "C Reference Manual,"

Bell Telephone Laboratories, Murray Hill, 1974.

- [10] D. M. Ritchie, "A Tour through the UNIX C Compiler," Bell Telephone Laboratories, Murray Hill, 1977.
- [11] 6800 PL/W Internal Specifications and Installation, Wintek Corp. May, 1976.
- [12] Jensen, K. and Wirth, N., PASCAL User Manual and Report, Springer-Verlag, Berlin, 1974.

INFORMATION PROCESSING NEEDS FOR SMALL BUSINESSES - A SURVEY

Robert E. Gibson

University of Delaware

Abstract

A survey was made of a number of local small retailers and professional firms. Information was gathered on their information processing needs. Results of the survey suggest that there are a large number of paperwork applications that could be performed quite efficiently on a microcomputer system. These include accounting functions such as accounts receivable, accounts payable, payroll, inventory, and general ledger. Other applications include client or customer file maintenance and word processing. Lower cost systems are becoming available, but education on the use and cost-effectiveness of computers is needed. In addition unknowledgeable users in small firms must be provided with all application programs and hardware maintenance and service.

Introduction

The development of the microprocessor chip and its rapid decline in price have spawned many applications in automotive, appliance, and industrial control. In addition it is responsible for the launching of the so-called Microcomputer Industry. This industry, now only about two years old, began as a mail order supplier of IC's, IC boards, peripheral equipment, and eventually complete microcomputer systems, primarily to hobbyists. Paralleling the growth of suppliers and the influx of new suppliers into microcomputers was the development of the retail computer store. The number of these stores has grown from a dozen or so in the U.S. 18 months ago to over 600 at the end of 1977. There are presently three existing franchise operations, and others are being formed monthly.

Most suppliers, franchisers, and store operators perceive the market for microcomputers in terms of its three major segments, the hobbyist, the home user, and the industrial and business user. The latter segment can be divided into (a) knowledgeable industrial OEM users and (b) the unknowledgeable small business and professional organization. This paper focuses on the unknowledgeable small business and professional market segment for microcomputers. In particular, the results of a survey of a small sample of small business and professional offices will be presented.

Development of the Market Survey Project

The project was developed and carried out by the students in the author's class in Marketing Research at the University of Delaware during the spring 1977 semester. The objective of the research was to explore information processing needs and the owner/manager's attitude toward automating information processing functions. A set of criteria was developed to guide in the selection of businesses and professionals to survey. These criteria included: organization too small to currently own their own mini- or large-scale computer system, inventories with a large number of items and/or rapid turnover, substantial client or customer list, substantial typing and documentation, need for frequent direct mail contact with customers or clients, and finally, large volume of paperwork and/or record keeping. A list of types of small businesses and professionals was prepared based on these criteria and each student selected a particular kind of business or professional endeavor to survey.

The reasons for this approach were twofold, first so that the students could select areas of interest to them to foster enthusiasm for the project, and second so that each student interviewer would rapidly become knowledgeable, if not expert, in his particular area and thus be able to more intelligently question the respondents and analyze the results. The retail businesses selected included Drug Store, Bicycle/Motorcycle Shop, Clothing Store, Record Shop/Radio Station, Bookstore, Jewelry Store, Liquor Store, Hardware Store, and Auto Parts Store. Professionals selected for survey included Physician, Attorney, Dentist, Surveyor/Engineer, and Insurance Agent.

Among the various interviewing techniques of mail, telephone, and personal, it was decided that the students would carry out the more extensive personal interviews on a small sample of firms. Thus each student was responsible for a minimum of twelve interviews. Separate questionnaires were developed for the businesses and the professionals. Due to limitations of student time and transportation, the universe from which the sample was selected was confined to New Castle County, Delaware.

Since the sampling process was not random, inferences of the results to a larger population are not anticipated. No claim is made even that the sample represents New Castle County, the base population. The results, interpreted as preliminary findings however, can establish a basis for more formal and extended research.

The Questionnaire

The questionnaire for businesses consisted of three major sections, basic information on the organization, information processing procedures and reports, and attitudes on computers. The basic information included name and type of organization, number of outlets, number of employees, number of years in operation, legal form of ownership, and approximate annual sales in dollars. The second section included questions on the type of information entered on sales slips and into the cash register. Additional inquiries were made into the type of credit offered, collection periods on in-house accounts, and time spent on updating client records. Detailed information was sought in this section on the type of management

reports, their frequency, and who within or outside the organization prepared them. Detailed information was gathered on the existence and use of customer lists, inventory content, inventory reordering policies, and average turnover. The amount of document preparation involving typing was also explored in this section. The final section of the questionnaire dealt with questions such as "What is your biggest paperwork headache?" and "Have you ever considered using a computer?", with reasons for and against computer use.

The questionnaire for professionals was similar in that it contained the same three major sections but the type of specific questions varied to accommodate the differences between a professional and a retail operation. For instance inventory was not a factor for the professionals but awareness of and need for word processing capability was significant. Questions related to the cash register were also deleted from the professional questionnaire.

The first draft of the questionnaire was tested by each student on two or three of his sample. Revisions were made to eliminate unnecessary, ambiguous, confusing, and over-sensitive questions. The elaborate coding scheme was eliminated since tabulation for the twelve completed questionnaires would be done manually by each student.

Selection of the sample organizations within each type was primarily a matter of accessibility. However certain guidelines were followed where possible. These included apportioning by size the number of organizations in the sample to approximate their ratios in the population. Balancing of the sample was also undertaken on the basis of city versus suburban location where geographic location was believed to be significant.

Analysis of the Basic Data

Basic data on some of the business organizations surveyed are shown in Exhibit 1. The median number of employees ranged from 1.5 for liquor stores up to 8 for hardware/building supply stores. Most store groups had a median of 3 to 5 employees. The largest store (hardware) had 32 employees. The number of firms incorporated ranged from 3 of 12 for liquor to 9 of 12 for

clothing stores. Most of the unincorporated were sole proprietorships; there were only a few partnerships. Annual sales had median values from \$150,000 to \$300,000 with some stores ranging up to \$2 million. The median value corresponds closely with the average retail store sales of \$180,000 in the United States.

Basic data on some of the professional organizations are shown in Exhibit 2. The median number of employees ranged from 2 for physicians to 9 for surveyor/engineers. Most firm groups had a median number of employees of 2 to 5. Two groups included relatively large organizations, attorneys ranging to 75 employees and surveyor/engineers to 120. The number incorporated ranged from 1 of 12 for attorneys to 9 of 12 for surveyor/engineers. Most of the unincorporated organizations were sole proprietorships. (Pretest of the professional questionnaire showed that income or revenue was perceived as too sensitive, so it was omitted from the final version.) One of the major factors affecting the volume of record keeping for the professionals was the number of clients, both active and total. Median number of active clients ranged from 20 for surveyor/engineers to 150 for physicians. Dentists followed physicians closely with a median value of 140. Total number of clients tended to correspond to active number of clients, with medians of 50 for surveyor/engineers to 3,000 for dentists. Both physicians and dentists ranged very high in terms of total numbers of clients with a high of 7,000 for dentists and 7,500 for physicians.

Potential Microcomputer Applications in Retail Stores

An open-ended question near the end of the questionnaire asked the respondent "What is your biggest paperwork headache?" Responses to this question revealed several common areas of concern. The most frequent "headache" was directly related to inventory control and included the reorder process, maintaining records of in-stock items, and paying supplier bills within the varying payment terms of each supplier in order to claim the prompt-payment discount. The inventory problem increased with the number of items carried -- running into five or six thousand for a hardware store -- and the number of suppliers regularly

utilized -- ranging to many hundreds for some stores.

The second largest potential microcomputer application for small retail outlets is that of maintaining customer lists, which can run as high as many thousands of names.

These lists are maintained for two general purposes, for customer billing where in-house charge accounts exist, and for direct mailings for promotional purposes. A special application was evident in drug stores, where federal and state laws require much record keeping with regard to prescription drugs. The patient profile is one such set of records which involves a complete history of drug purchases. Drug stores are also more likely to maintain in-house charge account systems.

Additional potential uses include keeping payroll records and printing pay checks, and integrating the various accounts such as receivables, payables, inventory, and payroll into a general ledger system.

Potential Microcomputer Applications in Professional Organizations

Although some common record keeping problems came to light in the survey of professionals, there were a number of differences also. Dentists and physicians reported similar paper headaches. The two most frequently mentioned were billing procedures and filling out insurance forms. Three of the ten dentists were presently using outside computer services for their billing, account aging, and general management information. An application that occupies a lot of time for dentists and physicians is that of recording patient records. An installed microcomputer system, if purchased for billing and insurance form purposes, could easily handle patient records and would likely be welcomed as more efficient and cost-effective. The persons now performing these tasks could then presumably focus on more urgent matters. A more sophisticated application mentioned by several physicians is that of computerized diagnosis. Such a medical information bank is now in development on an experimental basis at a major medical center.

Nearly all of the surveyor/engineer firms presently have a small computer.

These computers, with three exceptions, were used only for engineering calculations. The three firms used their computers also for billing and client transaction information.

Attorneys, as expected, were involved in a great amount of paperwork. Surprisingly, only one of the firms, the largest one with 75 employees, was using a computer. This firm made full use of the computer to assist in accounting, docket control, and word processing functions. It is believed that many of the smaller operations could effectively utilize a microcomputer system for keeping track of client records, court schedules, and for word processing use in the vast amount of document preparation typical of legal work.

Cost-Effectiveness of Microcomputers for Small Organizations

The only reference made to computers in the survey questionnaires consisted of an open-ended question at the end which asked if the respondent had ever considered a computer for his operation. Responses and reasons given centered around the perceived high cost and the bad image of the "impersonality" of the computer. There is a feeling that the personal store-customer relationship would be endangered by having a computer. An education process is needed here, first to show the capability of lower cost computers and secondly to demonstrate that the computer is simply a tool to permit greater efficiency and therefore greater profitability.

A typical microcomputer system now available for small business and professional firms would consist of a central processor unit with built-in power supply and interface boards. It

would have up to 48K words of internal memory and dual mini-floppy or standard size disk external memory units with about 180K or 600K capacity. It would include a low cost printer and a CRT or video input/output terminal. Such a system can be purchased at retail today for as low as \$5,000, an affordable cost to many small organizations.

We must add to the hardware cost, however, the costs of application software and of maintenance for the equipment. There are a few standard business programs now available. A typical one includes accounts receivable, accounts payable, payroll, inventory, and general ledger programs and costs about \$5,000 for a single-user license. It is believed that the cost of software will decrease significantly and, hopefully, the quality will improve. Maintenance and service are currently offered through local computer stores on the equipment that they sell. Since the use of microcomputers on a commercial basis is so new, there is little basis as yet for estimating the reliability of the hardware. A typical service contract for a \$5,000 system would cost \$50 to \$85 per month. Hourly service rates of \$20 to \$25 are also offered for equipment repair when needed.

The preliminary survey of small business and professionals described in this paper suggests that there are a great number of information processing applications for a low cost computer system. Recent computing and storage capability trends indicate that suitable computing and storage devices are available. Present cost trends suggest that the systems will be affordable. Major problems to focus on are those of user education, software development, and equipment reliability.

Type of Business	No. Employees Median (Range)	No. Incorporated / Total No.	Approx. Annual Sales (\$thousands) Median (Range)
Drug	3 (2-14)	8/12	150 (100-1000)
Jewelry	5 (1-19)	3/12	300 (100-1000)
Bookstore	3 (1-7)	5/12	100 (50-200)
Liquor	1.5 (1-2)	3/15	150 (100-1500)
Hardware Bldg. Supply	8 (3-32)	8/12	150 (200-2000)
Clothing	5 (1-20)	9/12	300 (150-1000)

Summary of Basic Data on Retail Stores
Exhibit 1

Type of Organization	No. Employees Median (Range)	No. Incorporated / Total No.	No. Clients per Week Served Median (Range)	Total No. Active Clients Median (Range)
Dentist	5 (2-25)	4/10	140 (50-550)	3000 (1000-7000)
Physician	2 (1-8)	4/10	150 (20-4000)	2000 (500-7500)
Surveyor/ Engineer	9 (2-120)	9/12	20 (2-35)	50 (10-300)
Attorney	5 (3-75)	1/12	50 (20-300)	200 (75-1800)

Summary of Basic Data on Professional Organizations
Exhibit 2

TRENDS IN WORD PROCESSING

Thomas J. Shinal

Vice President of Engineering

Computer Products Unlimited, Inc.

Gaithersburg, Maryland

Today's Text Editors are to Word Processing what pocket calculators are to accounting machines. Virtually all modern text editors employ from one to five microcomputers to manipulate data exclusively. It's a gross waste, considering today's technology, to limit these processors to single task functions when present business requirements demand an order of magnitude beyond text editors. It's tantamount to chipping an iceberg for an ice cube. The berg is today's word processor. The microcomputers are existent but lack the imagination and tools to use them. It is obvious that the tools (text editors) were developed to satisfy the task (editing) and were simply evolutions of earlier data terminals.

Now we evolve to today's technology. It has already been established that the tools exist. Our function now is to implement them into a versatile, usable system. Let's first upgrade the basic microcomputer into a computer that has the growth potential required for both foreseeable and unforeseeable future applications, within reason of course. Thus the constraints need not be severe. To get the power that we need, let's select a microcomputer that has a powerful software instruction set.

We want compatibility and speed as well as compatibility with a large computer. Compatibility to eliminate obsolescence. If our software is upward compatible and our peripheral equipment (printers, terminals, mass storage, etc.) are usable on larger computers, then we have achieved our goal. To frame an example, we'll select a processor manufactured Digital Equipment Corporation and build upon it. We could just as well have chosen Data General, Computer Automation, General Automation, Interdata, etc., but we'll stick to the example. The basic computer will be built into a text editor with sufficient memory, a CRT display terminal, a proportional full face multi-font printer and a powerful text editor.

Let's make some assumptions for a basic system:

1. Operation similar to a memory typewriter, i.e., no external memory in magnetic cards, cassettes, diskette, etc.

2. Cost - less than \$10,000.
3. Simple to operate, tutorial display.
4. Function Keys - no codes to memorize.
5. CRT display.
6. 45 C.P.S. printer with selectable, full-face fonts, proportional spacing.

This model allows text entry, complete editing, string search, cut/paste, justification, and printing. All in all, a fine machine for its intended immediate purpose. Now, if we've done our homework, we should be able to build this into a multi-function, multi-purpose, multi-user, general purpose computer which includes a powerful word processor.

The first enhancement will be to add storage. We'll take each option separately.

Floppy Disk

Most word processors available today employ a dual drive, single density system with a total of 512,000 characters of storage. The usual mode of transferring data to/from the floppy is via the programmed I/O mode (PIO). This is adequate in single task systems (dedicated text editors) but slow when we want more power in our system. (The processor must manipulate every character being transferred to/from memory)

The next enhancement would obviously be to speed this up. We do this by transferring disk data to/from memory by a mode called direct memory access (DMA). Now the processor can go directly to memory to get the data it needs only handling it once as opposed to many times. The actual throughput speed is increased manyfold. To the user, this reduces screen display updates from seconds to a fraction of a second. Now that we have fast data transfers, let's do something about the amount of data which can be stored.

The above system stored data in a single density format (256,000 characters/disk). This seems like a lot of data and it may well be for many applications. It amounts to approximately 68 pages (3,000 characters/page), less computer overhead. We lose more storage when we start to em-

play spooling. Spooling allows edited data to be stored as a separate file to be output to a printer while new editing takes place.

If you are preparing manuals, it becomes significant. It is always possible to store data on multiple diskettes but archiving and sorting can become a problem. Recent technology has now given us double-density (512,000 characters) of storage per diskette side and also double-sided diskettes. Now we have 1.2 million characters of storage per disk (1st quarter 1978) in the same space (2.4 million characters for a dual-disk system). This has other advantages which we will approach later.

Mini-floppy disk drives are showing up in more and more word processing systems. They are about 60% of the size of a standard floppy drive at about 75% of the cost with only 40% of the storage capacity. This does not offer any economic advantage. The only practical advantage is for systems which have a size constraint and require extreme compactness. The trade-offs must be justified by application. They will become more attractive only when capacity can be increased via the double-density-sided techniques. However, the pricing at that point will be on a par with standard sized, single-density floppy disk drives.

New trends in floppies lean not only toward the double-densities, double-sided, but towards redesign of the mechanics. Standard drives are approximately 3" wide and have head drive servomechanisms which use stepping motors. Now, dual drives are available which are only 4.5" wide and use "voice-coil" head positioning. It is not necessary to understand the intricacy of these techniques but only to realize that we can access data five times faster and that disk sizes have been substantially reduced. This again tends to diminish the need for mini-floppies.

Mass Media Disk

While diskettes afford convenient storage, limitations occur in capacity. Larger capacity is needed when we want to store multiple or large documents and additional programs, e.g., accounting, inventory control or other business, industrial or engineering applications.

Cartridge disk systems have become available within the last few months for small computers. Disk cartridges are removable media with storage capacities of 2.5, 5.0, and 10.0 million characters and provide dependable random access storage. This technology also employs direct memory access.

These disk drives are available in single or dual disk configurations. With recording densities of 100 or 200 tracks per inch and either 1500 or 2400 RPM per disk rotation. Single disk versions use IBM 5440 type removable disk cartridges. Dual disk units use the removable cartridge plus a non-removable fixed disk. At 1500 RPM the data transfer rate is 200,000 characters per second to the host processor. The 2400 RPM drives increase data transfers to 320,000 charac-

ters per second.

Physically, the packaging is slightly larger than the dual floppy disk system. The total cost about triples that of the dual floppy (double density, dual drive is about \$4,300) to approximately \$12,000 for a 10 million character storage system. The comparison, however, shows up as 0.36 cents/character (dual floppy) vs. 0.12 cents/character (cartridge disk). The cartridge costs about \$80 vs. less than \$8 for a diskette. Therefore, archiving cartridges becomes expensive.

A third alternative, which is just being made available today for small systems, is called fixed media disk. With this new hardware, we have all of the advantages of random access disk storage, high speed, DMA and storage capacities from 12 to 63 million characters, in a package slightly larger than the cartridge system at about the same cost. The cost of a 63 million character storage system is less than .02 cents per character. Winchester technology is employed to provide high storage capacity, very fast access and data transfer, exceptional reliability and immunity to external environments.

Three storage capacities of 12.5, 37.6 and 62.7 million characters are achieved with one, two or three disk platters. The media, recording heads and positioning carriage are packaged in an integral fixed module. This eliminates the costs and tolerances associated with removability. It also eliminates the need for head alignments and other periodic maintenance. The Winchester technology features low mass, high compliance recording heads which fly much closer to the recording surface than previous technologies would permit. Thus, they can read and write at higher data densities on tracks spaced much closer together. Data transfer rates of 7.08 million characters per second are achievable.

It is often necessary to provide a secondary storage media with removable media for archive purposes. Therefore, a combination floppy/mass media or cartridge/mass media system is not uncommon.

At this point we have invested about \$24,000 and have a powerful word processor with a proportional spaced printer and a mass memory storage system with floppy disk for archive storage. With all of these attributes, it seems somewhat costly for a single-user station. Now is when we can take advantage of all of this power. By adding software, such as a timesharing package and four additional CRT terminals at \$2,500 each, we arrive at a total cost of \$34,000 or approximately \$7,000 per station. Each station has complete access to all of the files in storage. By adding a financial software package, one user, or more, may elect to do payroll while others edit data.

The computer system's data entry functions permit the user to establish and change and/or delete a fill-in-the-blanks document form; to call up the form from the disk for filling in; and to apply various validity checks on filled-in data

before storing the completed forms in a file for transmission to a central computer for processing.

Optical Character Recognition

Optical character reader recognition (OCR) readers and printers are presently available to allow normal selectric typewriters with an OCR typeball to become data entry terminals. Copies may be "blue-line edited" for final copy on a CRT terminal.

Software is presently available to allow document printing functions that can cause a file to be formatted to permit the user a choice of font, format, page number, priority, and output a file to a magnetic tape or directly to a photo-composition system.

Magnetic Card

To allow compatibility with other systems, magnetic card readers are available which have a serial communications interface. With this device magnetic card archives may be purged and entered into other word processing systems.

Line Printers

Line printers, up to 600 lines per minute, are directly interfaceable to allow high speed printing of spooled data. However, the most popular printers used to date in word processing are those employing the daisy wheel developed by Diablo and emulated by Qume.

Although the speed is limited to 45 CPS or less, they have the advantage of crisp, legible characters which challenge the quality of the selectric typewriter from IBM. IBM has, in fact, contracted to purchase large quantities from Qume. There are added advantages such as changeable fonts, multiple fonts per carriage, and graphics capability.

Communications

Communications options include interfaces which connect word processing systems to Western Union TWX/Telex networks to automatically receive or send TWX messages or to send mailgrams and telgrams.

Software Languages

Software options include utilities such as batch mode which allows a sequence of operations to take place in an unattended mode. Languages supported for other software packages include basic; Fortran IV; APL; Macro Assembler; Focal; and "C".

Storage

Disk storage capacities have already increased to practical applications limits. What may be expected is that the 63 MB storage system will drop from approximately \$12,000 to about \$8,000 within the next 12 months. Processors

within the text editors will reduce in size by 50% as well as reducing in price by at least 30% by spring of 1978. Memories will see a price reduction of up to 30%. Our \$34,000, five station system will reduce to \$24,000 or less than \$5,000 per station down from \$7,000 by the fall of 1978.

The futures of floppy disk drives may be expected to wane as bubble memories take their place in the market. Bubble memories were developed by the Bell Telephone Laboratories a decade ago and are non-volatile when power is removed, much like magnetic core or floppy disk.

Magnetic bubbles are formed in thin sheets of certain magnetic oxides. Bubbles are shaped by applying a biasing magnetic field perpendicular to the plane of the sheet. Each bubble is a single wall domain in the shape of a right circular cylinder in the sheet. These domains can be moved from point to point in the two dimensional film structure. Their presence, absence or sense of magnetic direction at any location can be used to store information. Bubble memories transfer data at rates up to 1,000,000 bits/second. They are non-volatile but accessed serially (in order, one-at-a-time) so an index must be created to remove data.

Other forms of storage are showing promise and are known as "charge storage" devices. The "CCD" (charge coupled device) are "bucket brigade" type MOS devices which move a small quantity of electrical charge from one discrete location to another. A whole string of charges is moved at once which gives the name "bucket Brigade". A similar device called "EBAM" for electron beam accessed memory employs an unobstructed MOS chip on which electrical charges can be stored with an electron beam. The beam is non-volatile while the CCD is volatile.

At the present state of the art, CCD's are available in a form usable in computers; however, at this juncture, it appears that bubbles will appear first in large quantities.

Texas Instruments has recently introduced a printer employing bubble as a buffer in a terminal. The present costs prohibit its use as a replacement for floppy disks. A 20,000 character module presently costs about \$500. Consider that this would replace the diskette media at a present cost of under \$5.00 for a 256,000 character storage. The memory system initially would cost \$2-3,000 less than a floppy disk system and, as such, could stand a bubble module cost of \$25.00. This 20:1 cost reduction is provable but we may expect to wait a couple of years. More rapid price reductions may be precipitated by a competitor to T.I. or a few large orders insuring sufficient quantities to raise productions and yields.

Magnetic Bubble

VS.

Moving Head Disk

Advantages

Lower access time
Lower entry price
Smaller physically
Simpler interfacing

Disadvantages

Media removable
Higher bit price
Lower transfer rate

Floppy Disk

Lower access time
Lower entry price
Smaller physically
Simpler interfacing

Media removable

MOS RAM

Non-volatile
Lower bit price
More bits/chips

Higher access time
Complex interfacing
Lower transfer rate

CCD

Non-volatile
More bits/chips

High access time
Lower transfer rate

Only Intel, Rockwell and Texas Instruments seem to be engaged in bubble technology programs. Similarly, Fairchild and Motorola have been developing CCD's but have done little yet with magnetic bubbles.

Rotating disk memories employing Winchester technology logically will be mated with cartridge drives to provide archivable media both sharing a common controller for the cost of either system alone by today's prices.

"Hard" Display

The next peripheral to be upgraded is the display itself. Plasma displays are showing up but to date they have serious limitations by today's state of the art. Plasma, as used in the "plato" system, allows extensive graphics as well as the merging of microfilm display. The disadvantage is that the display is amber in color, which may be filtered to orange or red, and is harsh to the eyes over an extended period. Many people are color blind to this part of the spectrum. Some other "plasma" terminals are aesthetically small, however, limit their displays to less than 32 lines of 40 characters per line and, of course, retain their amber characteristics and unusually large characters.

I think that the approach to take with today's technology is to enhance the CRT display. CRT displays are presently segregated into two categories; the 25 line (Wang, DEC, CPU, etc.); and the 55 line (Vydec, Lexitron, Xerox, etc.). These camps are supported by the users who have a distinct preference for one or the other.

Those who favor the smaller display have the advantage of larger characters and white phospho-

rous. The smaller display presents a segment of the text for editing. The display then provides a window of 25 lines (or less) of memory from which the user may scroll through. Typically, the 25 line display employs a dot matrix for character generation while providing a legible display. It also has the capability of relatively easy modification for foreign fonts and special fonts, e.g., Library of Congress.

The users which prefer the 55 line display have the advantage of seeing a complete page on the CRT. They usually form characters by the stroke method, that is, a fully formed character rather than the dot matrix. While this forms a more legible character, the advantage is usually offset by the manufacturer as he usually reduces the character size by about 50%. The manufacturer who uses this display will typically present data in a page format only, i.e., if you overflow a page via editing, you will have to manually reformat all succeeding pages.

"Soft" Display

By combining the advantages of both methods, we may achieve a display capable of 55 lines which is selectable down to 12 lines for legible, easy editing much like a camera would employ a "Zoom" lens to focus on the details of interest. After the overall scene is examined, let's also retain the advantage of automatic page overflow of the small display so that subsequent pages are automatically reformatted on edit overflow.

For the character generation, we will retain the dot matrix; however, instead of the usual 5 x 7 matrix, let's increase the resolution to at least 10 x 12 and reduce the character size only when we display the full 55 lines.

There was a reason that we retained the dot matrix, other than economic reasons. It was briefly mentioned earlier that we would like multi-lingual capability. The advantages are obvious but let's also add one other dimension which encompasses all. "Multi-font". With this capability, we may now have underlines, italics and changes in pitch on the screen.

If your ultimate document is to be generated on a photo-composition system, wouldn't it be advantageous to see the display of data as close to final copy as possible? This is our goal! It's not out of the question or even impractical if we design our system properly. The basic display includes a default character generator which loads into RAM for display and "X" number of fonts which are stored on our system disk to be called into use as required.

Let's add one more feature to our "soft display"; the ability to build our own characters on the screen and load them into disk storage. We may use these special graphics to build forms on to the screen and/or to drive special output devices, e.g., custom printers or photo-composition systems.

Future

In the "Star Wars" category, we may expect the CRT and plasma display to give way to laser generated displays using a hybrid of both technologies. The CRT electron gun will yield to laser diodes (already priced in the \$30 range) used to ionize inert gases for plasma displays, or perhaps to use crystal deflection of the laser beams onto a phosphor screen. The technology lacking is that of economical, efficient, laser deflection. Can we also speculate a bit and let our display take the form of liquid crystal much like LCD digital wrist watches. Let's also use this crystalline network to store our data. Have you ever heard of holography? Is it possible to combine both? It does sound intriguing! Is this the flat TV that has been spoken about for the past 20 years? Look for it within five years.

A MODULAR OFFICE SYSTEM (MOS)

by
David J. Farber and Stephen H. Caine

Department of Electrical Engineering
University of Delaware
Newark, Delaware

Caine, Farber & Gordon, Inc.
Pasadena, California

This paper introduces the design of a new microprocessor-based personal computer terminal and its associated software systems. Together, they form the Modular Office System (MOS). The MOS is intended to bring together, in one design, the current LSI technology, the best current set of personal and inter-personal computer-based services, and a clean, easy to use human interface. It is not our intention to design with future technology but, rather, to design for currently (or soon to be) available technology in such a way that new developments can be smoothly integrated into the MOS. For this reason, we talk of CRT's¹ rather than liquid crystal displays, of INTEL 8080¹ or the HP MCC⁵ class microprocessors rather than those we know will be available in several years, and of floppy and micro-disks rather than bubble memories. In such cases, the above changes will either improve the price, the performance, or, in some cases, the capabilities of the MOS rather than create any functional new approach to the problem.

We realize that there are efforts in certain industrial research labs with similar goals. However, we feel many of these efforts are over-complicated and fail to appreciate either the forthcoming technology or the demands of users for simple extendible interfaces.

It is important to note that the system is not intended to serve programmers as programmers. Rather, it is intended for executives, designers, and other non-computing users of computers.

The MOS Terminal

The terminal, shown in Figure 1, incorporates sufficient computer power to support screen editors, message systems, and file systems. When appropriate, communication between the terminal and other modules in the office cluster can be supported.

The MOS Terminal (MOST) has a small (2 - 4 Megabyte) fixed micro-disk as well as a removable floppy-disk with a capacity of 750,000 - 1,000,000 bytes. This combination allows information to be removed from the MOST for privacy reasons or for transportation to non-connected MOS clusters.

There are two CRT areas on the MOST. One is used for text display in the conventional manner and will initially be a conventional tube

displaying at least 80 characters x 24 lines of upper/lower case with inverted video and blinking characters. The other, a much smaller unit, is incorporated into the keyboard and is used for labelling the soft keys which surround it and to display system status or changes in the user environment (e.g., "message waiting").

In addition to the soft keys, the MOST includes a standard alphanumeric typewriter style keyboard, a cursor control cluster (with an optional mouse), a HELP key to invoke system and subsystem help mechanisms, a RETURN key to cause exit from the current subsystem to the next higher level system and a RESET key to return directly to the highest level processor.

Both CRT displays allow inverted video, blinking fields, and protected areas. However, it is not expected that the user will "write" on the soft key display -- rather, he will observe it and hit appropriate keys.

Cluster Organization

The MOST is just one module of the typical MOS configuration. It is, however, completely self-sufficient and can be operated as a stand-alone unit. The main units (in addition to the MOST) of an MOS cluster are:

1. File Modules -- A file module acts as an overflow file system in an MOS cluster. It provides for archiving of information and supplies store-and-forward capabilities for the cluster. Also, certain external network activities may use the file module. A number of file modules may be attached to a cluster for capacity or reliability reasons.
2. Network Module -- This module handles all external communications between MOS clusters and other non-MOS systems as well as inter-MOS communications which do not go through the local cluster switch module data paths.
3. Functionally Specialized Modules -- These provide services which are not suited (either due to cost or complexity) for inclusion on each MOST. Typical specialized modules may include:

* Mathematical transformation modules

- * Elaborate text formatting modules (line filling, justification, pagination, etc.)
- * Printer modules (line printers, typesetters, etc.)

All modules (including the MOSTs) of an MOS cluster are interconnected by a communications switch. The details of the switch are discussed later in this report. Basically, the switch acts as a telephone exchange connecting, on demand, the various MOS modules.

Operational Overview

When a user activates a MOST, he mounts a floppy which contains that portion of the MOST software that personalizes the unit for the user. This personalization may include specialized labeling of the soft keys, a private directory of names to be used by the message system (nicknames), and, perhaps, specialized processing modules. It is not intended that all of a user's data reside on floppies. Much of the data base will be on the micro-disk and on file modules.

It should be noted that unprocessed mail may be placed on a floppy prior to leaving the office; taken home to a stand-alone MOST; processed there to create new mail to be sent (which will be temporarily stored on the floppy); and returned to the office for transmission. On startup, the MOST will process any mail to be sent which resides on the floppy. This also provides the mechanism for stand-alone operation in the event of system failure.

On completion of startup, a set of functions will be displayed on the soft key CRT. Status information will also be displayed. Status will include such information as "message waiting," current date, and current time of day. Date and time will be supplied from a self-contained clock chip which will be able to continue operation in the absence of line power.

The user can now direct the operation of his MOST primarily via the soft keys. Typical software processors will include:

- * A mail reader, with reply and forward capabilities
- * A mail composer/sender
- * A file editor (screen based)
- * A reminder system
- * A calendar system
- * A calculator system

We feel that the MOS and the MOST form a highly flexible system capable of being economically built with current technology and capable of evolving with future advances in the technology.

The MOS Communications Section

The basic concept behind the communications section of the MOS system architecture is to allow the interconnection of MOS terminals with each other and with service modules of an MOS cluster via a name addressed virtual circuit digital switch. While the technology underlying the switch is not important to the operation of the MOS system, it is worth discussing the alternatives allowed by existing technology and to show future directions. A more complete description of the communications system is found in references [1] and [2]. Also, it is important to note that the hardware architecture described below is just one of a class of allowable schemes. Rings, Ethernets, etc. are also usable although possibly overkill for this application. It should be noted that in certain environments, a conventional telephone system could be used in place of or in addition to the MOS communication system. It is also possible to attach remote MOSTs to the system via the Telco connection module (see Figure 2).

The switch unit is initially "dialed" by a requesting unit with the name of the terminating unit. In the case of MOS terminals that is the name of the person who "owns" the terminal. The general sequence of actions is as follows:

1. The requesting unit software examines the message sent to it internally by the sending task. The name of the destination is examined relative to the local directory of names used by the owner vs MOS cluster names. Any translation is made prior to connection with the switch unit.
2. The requesting unit sends over a command line the destination unit name. The control unit of the switch examines its directory to see what the total name is of the destination. This will allow the use of simple names to be expanded into complex routing directions for internetwork routing.
3. The switch makes a circuit connection with the destination unit. During the course of this attempt a busy might be found which will be sent back to the sender or a sending process for his action. If a circuit can be opened a full duplex (logically speaking) path is established and maintained until either the sender or receiver "disconnects."

The action of the switch is very similar to that of a telephone exchange and can be viewed as a digital PABX. It is the intention of this design to allow the use of such systems as they become available in the near future. In such systems, the present technology suggests that a 6 wire (3 pair) connection exists from each unit to the local switch (it is noted that digital PABXs are by their nature distributable for efficiency and reliability). This would give two 64,000 bit per second data paths and a full duplex 2400 bit path for commands and signals. Thus, in the near future the MOS system could use the same

communications system as the forthcoming digitalized voice PABX systems and thus realize the cost advantages of piggybacking on the vastly bigger telephone market. During the interim, till digital PABXs arrive and in those places where existing telephone systems preclude their replacement, one has two paths. One is to use some local communications system such as an LNI Ring or an Ethernet structure; the other and perhaps preferable is to construct a simple MOS version of the data PABX.

Terminal Software System

The architecture of the terminal software is very simple, being based on an available message-driven real time system. An example of such a system is the Intel Real Time Monitor (RMX).^{3,4} RMX provides for an arbitrary number of tasks. Inter-task communication and synchronization is provided by sending "messages" to "exchanges." The primitives include SEND (a message), WAIT (for a message), and ACCEPT (a message), as well as those for creating, suspending, and destroying tasks.

The tasks which provide basic terminal functions operate under the RMX. These various tasks will include:

1. Micro-disk handler
2. Floppy handler
3. Keyboard handler
4. Main CRT handler
5. Soft-key handler
6. Soft-key CRT handler
7. Cluster communications handler

The RMX and the basic tasks reside in ROM, thus leaving the main RAM for use by applications and data.

The applications programs reside on the micro-disk and are brought into RAM on request. Since each terminal is actually a single-user system, there appears to be no need for fancy, paged, memory management systems. When one application calls for the execution of another as a subprocess, the context of the invoking process is saved and the new process is loaded in place of the old one. If necessary, any purely local data of the invoking process can be rolled out to the micro-disk. When a process terminates, its invoker (and the invoker's local data, if any) is brought back in, its context is restored, and its execution is resumed.

References

- [1] Farber, D.J. A Note on the Design of a Local High Speed Distributed Circuit Switch, Caine, Farber & Gordon, Inc. Discussion Paper Nr. 7708-2.
- [2] Farber, D.J. On the Control of a Distributed Circuit Switch, Caine, Farber & Gordon, Inc. Discussion Paper Nr. 7708-3.
- [3] Intel Corporation Data Sheet on RMX.
- [4] Intel Corporation PL/M-80 Reference Manual.
- [5] Hewlett-Packard Corporation MCC Specification Sheets.

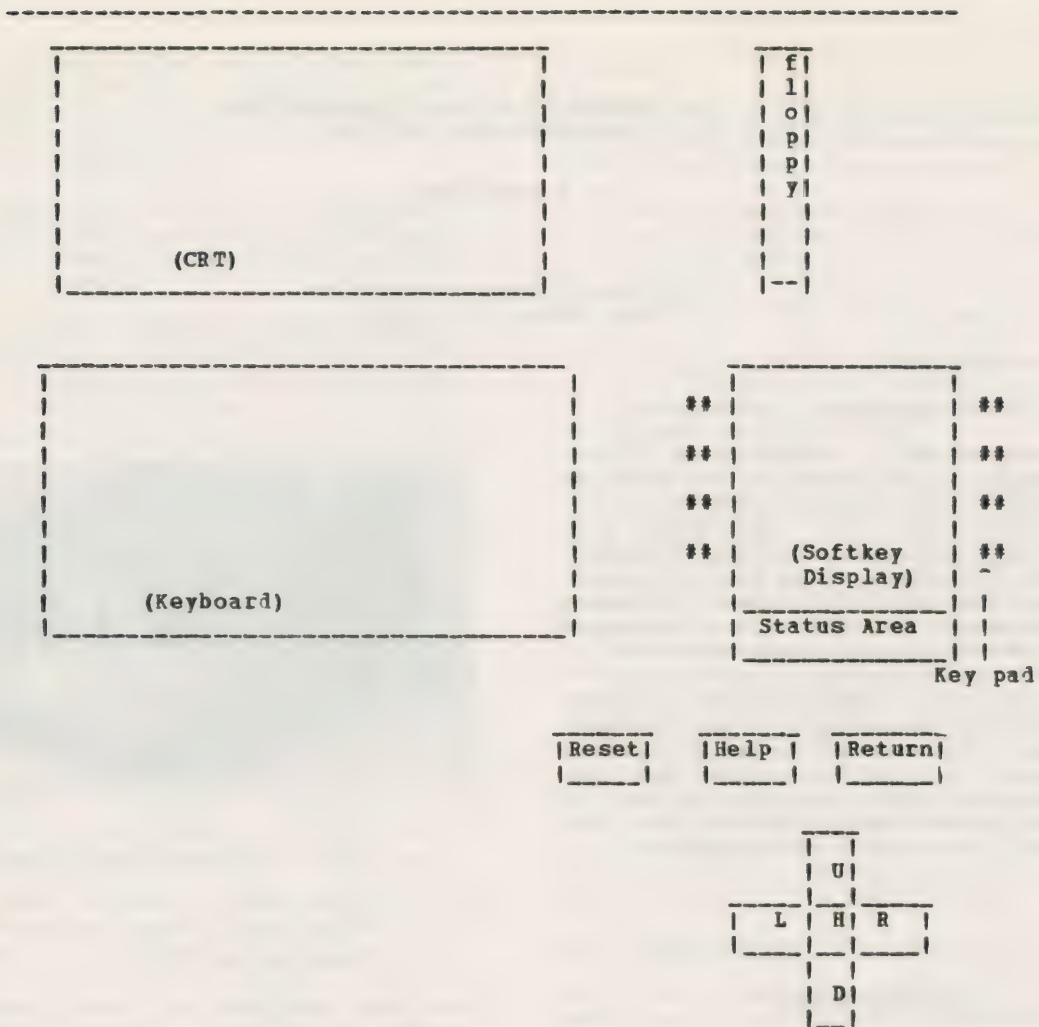


Figure 1

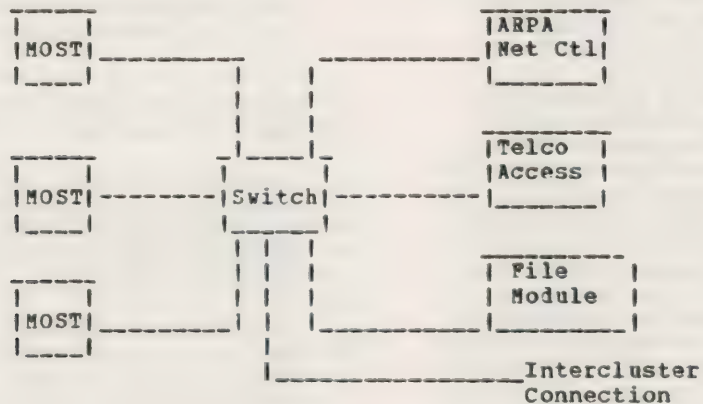


Figure 2. An MOS Cluster

The Design of A Small Interactive Computer--THE IBM 5100

D A Roberson

IBM General Systems Division
Boca Raton, Florida

Abstract

The advent of small, interactive, high-level language computers is the product of numerous recent architectural and technological advances. This paper presents a detailed discussion of a number of these advances as they apply to the design of the IBM 5100 system. Through this discussion, a rationale is presented for the decisions made and the trade-offs that were involved in the selection and the development of the various components in the system. This material is preceded by an overview of the IBM 5100 itself to give the context for the discussions of the system design and followed by some considerations for the design of future desk-top computers.



Figure 1 The IBM 5100 portable computer

Introduction

Every computer project, if it is to be successful, should begin with a single, fundamental design goal to serve as the guiding principle for the architecture, design and implementation of the machine or system. For the IBM 5100, this goal was the development of a "personal computer," a computer for the noncomputer professional to use in the solution of his or her technical problems. To achieve this design goal, a computer should be both easy to use and easy to learn how to use, yet it should provide a high level of function to allow for the simple solution of complex problems. To be "personal," it follows that the computer should be available when and where it was needed, and it should provide a measure of privacy and data security. Finally, the machine must be affordable for a large group of professional problem solvers. The embodiment of our design goal and the resulting design tenets is a portable standalone computer which supports interactive processing at the APL and BASIC language levels.

Overview of 5100

The base 5100/5110 is a small portable package containing a general purpose keyboard, a CRT display and an external CRT monitor jack, and either the APL or BASIC programming language with 16K bytes of working storage (see Figure 1). Features available with the unit include:

- 1) The language not selected in the base unit to provide dual language facilities.
- 2) Up to three additional 16K byte increments of storage.
- 3) An 80 or 120 CPS, 132 character per line, wire matrix printer with an adjustable forms tractor, multipart forms capability and a fine point plotting option.
- 4) One to four external one megabyte diskette drives to provide a relatively large on-line storage area, a sort, merge capability for data files, and to provide a medium for data transfer from one system to another.

- 5) An internal and/or an external tape drive feature to allow portable operation of the system with one tape and two tape operations for tape sort, merge applications.
- 6) A communications feature which allows the unit to function as a terminal, using either the start/stop or Bisync line discipline.
- 7) A serial I/O adapter feature which allows the attachment of a wide range of instrumentation and I/O devices. This feature provides an EIA RS-232-interface² with user-selectable line rates from 20 to 9600 bit/s and user-selectable data representations from 5, 6, 7 and 8-bit codes allowing attachment of devices with Baudot, EBCD, ASCII, and 8-bit serial data interfaces.
- 8) A parallel I/O adapter feature which allows attachment of the growing number of I/O and instrumentation devices which support the IEEE-488-1975 bus standard.
- 9) A range of tested program packages for statistics, economics, mathematics, graphics, and business and finance are available on tape cartridges.

Looking into the base system components in a little greater depth, we observe the following:

Keyboard

The keyboard is a primary input device into the 5100/5110. As such, it performs the following functions:

- 1) The entry of user programs and data.
- 2) The entry of System Commands and BASIC language keywords.
- 3) The direct control of the primary output devices, the display, printer, diskette drives and/or tape drives.

To accommodate the first functional requirement, the keyboard is composed of a standard typewriter-like keyboard with a four-function calculator pad concatenated on its right-hand side. The second and third functions are provided by additional CRT/cursor control keys clustered above the calculator pad and by a command key function. This command key function allows 5100 system and I/O commands and BASIC keywords to be entered with a single command shifted keystroke.

Display

The display is the primary interactive output device for the computer. Its principle functions are to provide:

- 1) Immediate visual input verification and the ability to easily alter incorrect input;
- 2) Prompting for command and data entry;
- 3) A display of the intermediate and final results for numeric and non-numeric calculations and other programming inputs and;
- 4) A display of programming and system error messages.

To provide this function, a compact 7.7 x 10.3 cm (3 x 4 inch) cathode ray tube display is imbedded in the portable computer. The display presents the user with up to 16 lines by 64 characters of messages and data. In conjunction with the keyboard, the display provides a variety of editing functions including character replacement, in-line insertion and/or deletion and character line scrolling (up or down) for in-line updating and re-execution. Finally, a display HOLD function allows the user to temporarily interrupt the output of data to allow for detailed examination of a portion of the data being output.

Tape and Diskette Cartridges and Drives

These units serve as the machine readable, non-volatile storage for the system. As such, they must provide the following functions:

- 1) The 'save' and 'restore' functions for a user's programs and data.
- 2) The cartridges must serve as the interchange media for transferring programs from one system to another.
- 3) The drives must serve as the system file for routines requiring a larger storage than that available in the main storage of the machine.
- 4) They must contain the special diagnostic routines used to perform a thorough test of the system.³

To perform these functions, the 5100/5110 contains a compact, low-cost tape drive for the IBM data cartridge, or attaches an external floppy diskette drive. The tape drive allows the computer to read data at 2850 characters per second and write with

read-after-write checking at 950 characters per second. The total capacity of the cartridge is somewhat dependent on the user's file format, but the capacity should exceed 200K bytes on the 300-foot tape.

The diskette drives allow the system to read or write data on the standard 8" floppy media with any of the three IBM recording formats and densities (i.e. 256KB on a single side, 512KB on two sides, or double density 1MB on two sides).

Languages

As was pointed out in the design principles in the introduction, the languages must be easy to learn, easy to use and must provide a high level of function. These principles dictate the need for a high level interactive language as the base machine language.

The two interactive languages selected for the 5100 were APL (A Programming Language)⁴ and BASIC (Beginners All-Purpose Symbolic Instruction Code).⁵ Beyond satisfying the design principles, APL and BASIC are two of the most popular interactive languages with numerous users and program libraries available for each.

The hardware architecture of the 5100 is organized around the two-bus structure of its microprocessor (see Figure 2).

Storage and Cycle Steal Bus

The storage and cycle steal bus emerges from the right side of the microprocessor and extends to the executable read-only storage (ROS), control store, the read/write storage, and the display adapter. The executable ROS contains the I/O supervisor and control routines, diagnostic and bringup routines, and the BASIC and/or APL microprogram sub-routines.

The read/write storage is composed of medium speed (≈ 300 ns access/450ns cycle time) MOSFET chips organized in 8K x 18 storage increments. This storage is used primarily as the work area for the user's programs. Additional uses for the read/write storage space include storage for the microprocessor's register space, storage for system tables and constants, I/O buffer space, and the 1024-byte display buffer.

The 1K byte buffer is used as follows: after characters of data have been stored in the buffer by the microprocessor, the display adapter cycle steals the data to temporary storage on the adapter card. The adapter card then uses an internal ROS character generator and a parallel load shift register to translate and serialize the data for presentation on the CRT display. The display must be continually refreshed, hence the adapter is continuously stealing cycles from the microprocessor for as long as characters are being presented on the display.

I/O Bus

Left of the microprocessor (see Figure 2), the program controlled I/O bus is directly connected to the base I/O adapter. This adapter contains the microprocessor interface logic for the internal tape drive, the keyboard and the operator control panel. In addition, the base I/O adapter performs an initial parity check on the data bus out and on the device address bus. These buses, along with the system clocks and control signals, are then repowered for distribution to the nonexecutable ROS adapter, and if present, to the communications adapter, the serial I/O adapter, and to the external I/O adapter.

When selected by the device address bus, the nonexecutable ROS adapter uses information on the bus out and the control signals to generate the required address and control signals to access microinstructions and data from the APL or BASIC ROS. The adapter then puts the requested microinstruction or data on the bus in for use by the microprocessor. This base function of the adapter is enhanced by the inclusion of a microprocessor loadable and readable ROS address register with autoincrementing capabilities. This register coupled with a buffer to the bus allows instruction and data fetch operations to be overlapped with the microprocessor's operation. This technique essentially eliminated the performance penalty which otherwise might have come with our use of a low-speed (≈ 2 microsecond access), high density 48K bit/chip n-channel MOSFET ROS technology. The large capacity implied by this technology was itself one of the key elements in allowing us to provide the dual language function available on the 5100.

The start/stop communications adapter and the serial I/O adapter both perform the similar function of interfacing the microprocessor's I/O bus to the EIA RS-232-C line standard. The interfacing function includes a parallel-to-serial conversion of data from the byte-wide bus out to the bit serial external standard, and a serial-to-parallel conversion on the data entering the 5100 to be put on the bus in. In addition, the signals are driven and received at EIA RS-232-C levels and timed to match the RS-232-C standard.

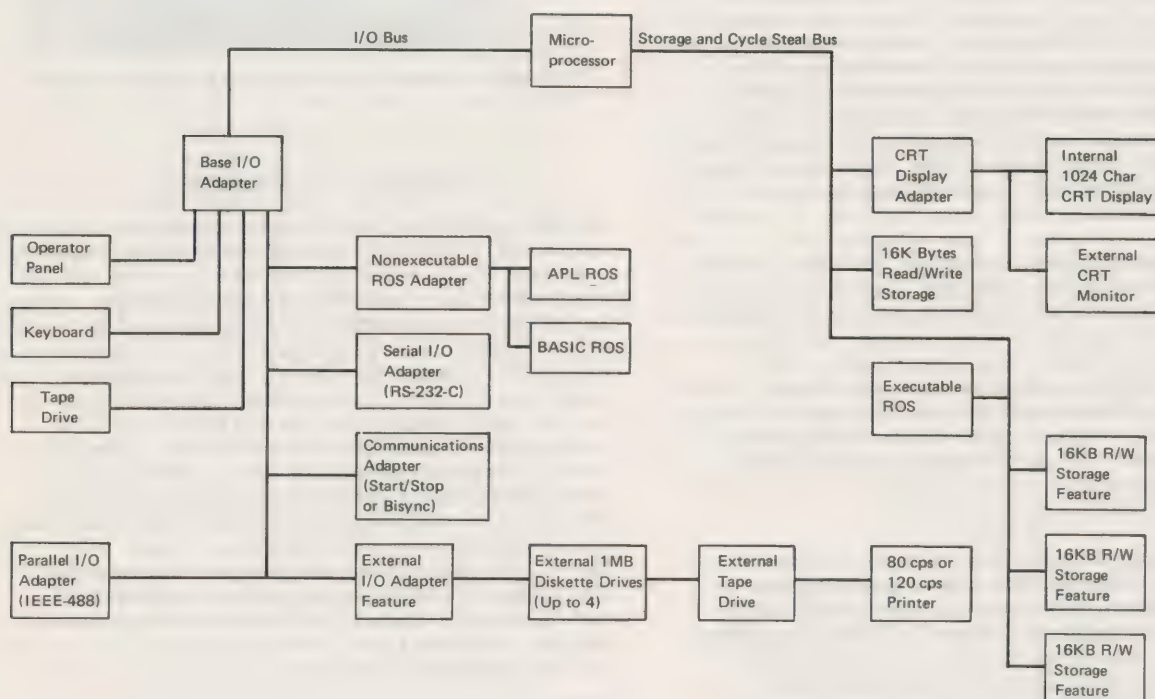


Figure 2 5100/5110 internal organization

The binary synchronous communications adapter also performs the parallel/serial conversions, but in addition it contains local intelligence to allow it to handle much of the bisync line protocol. This local intelligence not only off-loads the main microprocessor, but with its associated buffers, it allows much higher performance and function for the communication channel. From a host standpoint, the 5110 bisync communications is compatible with the IBM 3741 or 2770 in line protocol and has line rates up to 4800 bps.

Additional capabilities of the feature include auto answer, multipoint operation, and an optional integrated 1200 bps modem.

The Parallel I/O Adapter translates the internal bus into the proper lines for the IEEE-488-1975 interface. This interface provides for parallel data transfer from up to 14 different external devices.

The external I/O adapter interfaces the internal I/O bus to the external I/O bus providing drive for each of the various output signals. The printer, diskette drive and external tape adapters each provide the special logic required to interface the external I/O bus to its device. In addition, these adapters along with the other I/O adapters each perform parity check functions on the device address bus and the bus out and generate parity for the bus in. In this way, if there is a

loose connection or component failure anywhere within the computer, there is an extremely high probability that it will be discovered and isolated quickly.

Microprocessor

Returning to the 5100's heart, its single card microprocessor (see Figure 3), we observe that the microprocessor performs both the language processing function and the I/O control task. To perform these tasks the microprocessor instruction set includes control, device I/O, storage fetch and store, register move, arithmetic, logical, immediate, and jump (or skip) microinstructions. Mnemonic-wise, the sixteen bit microinstructions have opcodes and modifiers to provide a total of 49 separate microinstruction mnemonics. The average execution time for the microinstruction set is approximately 1.75 microseconds.

The addressing capabilities for the microprocessor includes the ability to address up to 64K bytes of read/write main store, 64K bytes of executable control store, (increased to 96KB for the 5110), 64K bytes of BASIC nonexecutable control store and 128K bytes of APL nonexecutable control store. The microprocessor also provides 16 device addresses with up to 64 sub-device addresses under each device address.

Finally, the microprocessor can address 64 eighteen-bit local storage registers which are also addressable as the first 128 bytes of main storage. This register space is further divided into four 16-register spaces, one for each of the microprocessor's four interrupt levels. With this interrupt level selection of a set of registers, the first register of each level can be made to serve as the microinstruction address register for that level. This structure implies that a complete task switch can occur at any time with all registers in the interrupted level immediately preserved and all registers of the entered level immediately available.

The microprocessor has two basic data path widths, one byte and two bytes (where a byte is considered to be 8 bits of data and one parity bit). The microprocessor passes data to and from executable control store and the read/write storage on a pair of unidirectional, two-byte buses. The I/O bus in and bus out paths on the other hand are only one byte wide. The ALU of the microprocessor is one byte wide with a carry added to the high byte.

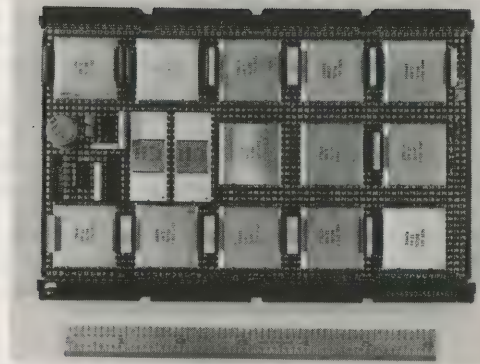


Figure 3 The 5100 microprocessor card

Firmware Architecture

The firmware or microcode organization of the 5100 may be divided into a language-related microcode organization and an I/O microcode structure. Stated simply, the language microcode must analyze the instructions and data received at the APL and BASIC language level and generate, from this, sets of related low-level instructions which are executable on the hardware of the microprocessor. The low-level results obtained must then be translated back into a character format which is recognizable to the user. Through the initial analysis process, the high level instructions must also be checked for their conformance to the syntactic and semantic rules of the language returning an informative error message if any of the rules were violated.

Once the functional definition of a language has been established, there are two primary indicators of merit for the interpreter which correctly implements the definition. These are the performance of the language in executing the user's routines and the storage space required by the interpreter. One way of reducing both the interpreter storage requirement and the development time without significantly degrading the language performance is to emulate an intermediate architecture and write the interpreter in this language.⁶ This was the approach taken in the development of the APL and BASIC interpreters for the 5100.

Turning to the I/O device structure, it is important to begin by considering the I/O device attachment philosophy. This philosophy was to put as much of the device attachment logic as possible into microcode. The advantages of this approach -- given the availability of high density, low-cost ROS -- include reductions in the

size, weight, power consumption, and cost of the attachments. These advantages are of course very important in the development of a low-cost, portable computer.

The potential disadvantages of the approach include the possibility of either poor performance for I/O operations or poor language processing performance due to the I/O burden or a performance degradation in both the processing and I/O areas. For a personal interactive computer like the 5100, the impact of these potential disadvantages can be essentially eliminated. An observation which supports this contention is the fact that for a personal or small business computer, the user normally performs most operations serially. Therefore, a personal computer is only required to provide limited facilities for overlapping I/O operations or for overlapping processing and I/O. This small limitation significantly reduces both the burden on the microprocessor and the complexity of the device attachment.

In order to provide good I/O performance while maintaining the "smart microcode/dumb hardware" attachment philosophy, the microprocessor itself should possess the following characteristics:

- 1) The microprocessor should be comparatively fast; that is, the average instruction time and interrupt response time should be low.
- 2) The microinstruction set must be relatively rich, particularly in I/O related instructions.
- 3) The microprocessor should present a large enough number of control and data lines to the device attachment to minimize the amount of required attachment logic.

To fulfill the language processing requirements of the system, high performance and instruction set richness (points one and two above) are both important for the microprocessor. In addition, a good storage interface with flexible addressing modes is an important attribute. These desired characteristics strongly influenced the design of the microprocessor in the 5100 (described in the previous section).

Summary and Future Considerations

In the selection and/or design of each of the machine's hardware and firmware components, our design goal and tenets were used as a basis for determining the proper course. As with all projects, the 5100

development faced normal business constraints in both resources and development scheduling.

Looking to the future of personal computers there are a number of advances which should provide keys to understanding the next generation of small machines. These include increased packing density of circuits per square centimeter, and the microprocessor evolution to smaller and more powerful components. These advances should reduce the physical size and weight of the units, enhance the function, and further improve the reliability of small computers.

From the user's viewpoint, the small computers will probably become even easier to use. This will be accomplished primarily by improvements in the language and the human interface, the keyboard, the display and the personal media. The language improvements should make them more interactive, easier to understand, and more powerful. Primary storage management will continue to be invisible and the management of secondary storage will become even easier.

Beyond these general directions, the specifics should make the computer area an exciting part of the world of computers for the years to come.

References

- 1) D. A. Roberson, "An Architecture for a Small Interactive Computer -- The IBM 5100", Proceedings of Compcon Spring 1977, pp136-139, March 1977.
- 2) D. A. Roberson, "A Microprocessor-Based Portable Computer: The IBM 5100", Proceedings of the IEEE, Volume 64, Number 6, pp 994-999, June 1976.
- 3) EIA Standard RS-232-C, Washington, DC, Electrical Industries Association, August 1969.
- 4) IBM 5100 Maintenance Information Manual, IBM order number SY31-0405.
- 5) K. E. Iverson, "A Programming Language", New York: Wiley, 1962.
- 6) J. G. Kemeny and T. E. Kurtz, "Basic Programming", New York: Wiley, 1967.
- 7) P. A. Smith and H. J. Meyers, "Personal Language Directed Systems, Theory and Practice", Proceedings of COMPCON Spring, February 1976.

AN APPROACH TO MICROCOMPUTING WITH BIT-SLICE MICROPROCESSOR

Richard J. Smith
Steven L. Clapper

RCA, Missile and Surface Radar
Moorestown, NJ 08057

Abstract

A gap exists between the bit-slice microprocessor as a chip and the actual realization of hardware and firmware which is based on these devices. LSI microprocessors represent complex subsystems, but they do not stand alone and require supporting hardware and software to complete the realization of bit-slice microprocessor based systems. This paper describes an approach for the operational use of bit-slice microprocessors.

The basic hardware elements required for implementing bit-slice microcomputer control are described. These include the register arithmetic logic unit (RALU) which is also called the bit-slice microprocessor, the microprogram controller, microprogram memory and macroprogram memory. Software topics include a summary of the approach taken for software development.

Basic Microcomputer Hardware Elements

Figure 1 shows the basic elements of a microcomputing system: a microprocessor register arithmetic logic unit (RALU); a macro-program, micro-program, microprogram controller, data RAM, and I/O function. This paper is concerned primarily with the basic computer control loop consisting of RALU, macro-program, microprogram controller and micro-program.

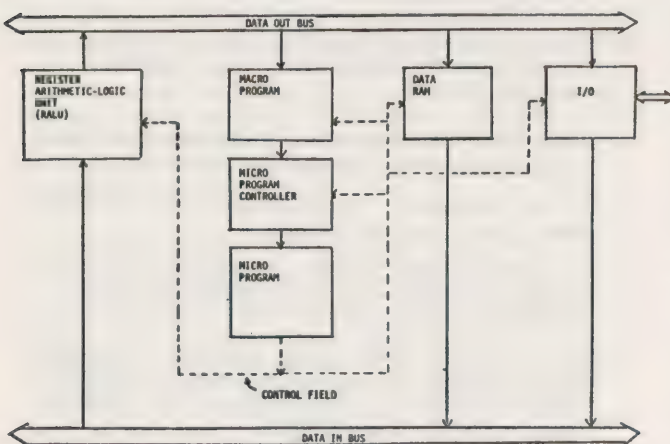


Figure 1. Microcomputer System

The RALU is the fundamental computing element in the microcomputing system. Figure 2 is a simple block diagram of an RALU. The term "bit-slice microprocessor" has been applied to this computing element by some semiconductor manufacturers. Bit-slice microprocessor means an RALU (usually 4 bits) which can be cascaded to form RALU functions with larger word sizes.

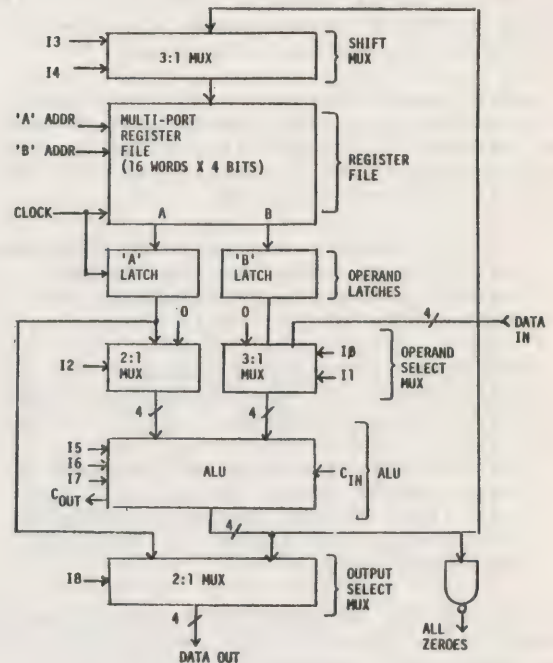


Figure 2. Register Arithmetic Unit

An RALU consists mainly of what its name implies: a register file and an arithmetic logic unit. The arithmetic logic unit performs arithmetic and logic operations on data supplied from several possible sources. The main source for arithmetic/logic unit arguments is the register file. The RALU is a latched device, which means that when the clock is low, data read from the register file will change as the register file address changes, however, when the clock is high, the register file data is latched and held stable. A complete RALU cycle is completed during one low-high clock cycle called a microcycle. The basic operation performed by the RALU within one micro-

cycle is:

- Read two operands from register file.
- Perform ALU operation on selected operands.
- Perform right shift, left shift or no shift on ALU output result.
- Write result back to register file.

This means the fundamental computer operation of reading a source ('A') and destination ('B') register, performing an elementary operation on the arguments, shifting right or left and writing the result back to the destination ('B') register as described by:

$A+B \rightarrow B$ or $(A+B)/2 \rightarrow B$ or $(A+B)*2 \rightarrow B$

is performed in one microcycle. A typical time for this function for 16 bits is 250 nsec.

This type of RALU or bit-slice processor as described above is available as a single LSI device and one of the most popular of these devices is the AMD-2901. The AMD-2901, a bipolar device (low power Schottky), is multiply sourced and is the microprocessor discussed in this paper.

Although the AMD-2901 is called a microprocessor it performs only very elementary functions (add, subtract, AND, OR, etc.) in any one clock cycle. More complex functions can be performed by combinations of the elementary 2901 functions. This is done by doing sequential elementary 2901 operations and adding peripheral logic around the 2901. One way to achieve this is by coding a sequence of 2901 instructions in a ROM and sequencing through combinations of 2901 operations to perform more complex functions. This sequence of control bits is changed at the microcycle rate and is called the microprogram. In addition to controlling 2901 operations, the microprogram ROM will contain control bits for other functions found in a microcomputer system.

One of the key features of a microcomputer system that distinguishes it from a random logic design is the ability to make decisions. Decisions are usually made based on a result: equal to zero; not equal to zero; overflowed; negative or positive; and presence of a carry out. These decisions will be made using flags from the 2901. These flags are: all zeroes, overflow, carry out, and sign. The all zeroes is provided by tying the outputs of the all zeroes outputs from each four bit slice together. The sign flag is provided by simply monitoring the most significant bit (MSB) of the data. A high (1) indicates a negative value. An overflow is indicated when the 16 bit two's complement value exceeds its boundaries. A carry is not an overflow and may occur anytime. The carry flag is used to indicate a carry and would be used if double precision arithmetic were used (i.e., 32 bit precision with 16 bit hardware).

In the microcomputer system described in this paper, microprogram bits are used quite freely and consequently the microprogram control word can be rather wide (i.e., 80 bits). Microprograms, although

requiring a wide control field, do not usually require a deep field and a typical microprogram ROM might be 512 words by 80 bits.

The capability to perform decisions in the microcomputer can be obtained by partitioning the microprogram ROM into two separate memories (i.e., 512 words partitioned into two 256 word memories) and selecting between them as a function of the state of some flag (i.e., all zeroes, sign, etc.). Figure 3 shows a typical microprogram ROM configuration. Both ROMs in the figure decode the same 8 bit address simultaneously, however, only one ROM (as a function of the memory select line) is enabled. The outputs of the ROM's are tri-state and are tied together, but only the outputs of the enabled ROM are clocked into the microprogram registers. The microprogram ROM words contain control bits for directing RALU operations and for controlling other elements in the microcomputer system. These microprogram ROM words are called microinstructions. Microinstructions from the selected ROM are clocked into the microinstruction register and held for a full microcycle.

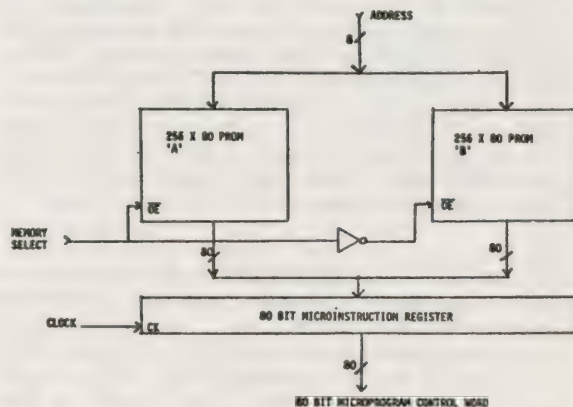


Figure 3. Microprogram ROM Configuration

The microinstruction contains several fields. For example, several bits will be allocated to the 2901 microinstruction while others will go to other elements within the microcomputer system. These control bits are usually ANDed with the clock to load peripheral registers. This technique allows the microprogram to strobe various registers as a function of the control bits while maintaining synchronization within the microcomputer system.

The memory select line shown in Figure 3 is used to conditionally select one microprogram ROM or the other. The logic which translates the possible decision conditions (i.e., sign, overflow, etc.) to the one memory select line, as well as the logic to address and sequence microinstructions is contained in the microprogram control function.

The microprogram is addressed by a function called the microprogram controller. The microprogram controller can represent the most complex function in a microcomputer system. The AM-2909

is a microprogram controller designed to be used with the 2901. It is also a four bit "slice" and may be configured for large microprograms. The output of the microprogram controller is a microprogram address and for the microcomputer system described here eight microprogram address bits are required (eight bits address plus memory select line required for 512 words of microprogram). Figure 4 is a block diagram of two AM-2909's configured for eight bits of microprogram address. An opcode (from the macroprogram) is loaded into the 2909 register during a microfetch cycle (RE LOW). The microprogram address (8 bits) is selected during any microcycle from one of four possible inputs. A direct jump input, the address register, the register file or a counter called the microprogram counter. The address register will always contain the opcode since a separate counter register is incorporated. The input to the counter register is the last microprogram address which can be incremented in the incrementer (CN HIGH). The address register, multiplexer, incrementer, and microprogram counter register perform the basic microcontroller function of loading a microsequence start address and counting. The 2909, however, has more capability than just the basic functions. The multiplexer, for example, allows the microprogram address to be selected from direct inputs. This means that a jump address can be incorporated in the microinstruction. Another powerful feature of the 2909 is the capability to do microsubroutines. This is accomplished by pushing the current microprogram value onto a four word stack (register file). The stack pointer is automatically maintained. Control of the stack is via the P/P (push/pop) line and FE (file enable) line. When a jump to microsubroutine call is executed, the return microprogram address is pushed on the stack and the microprogram branches to the microsubroutine. At the completion of the microsubroutine a return is initiated which pops the return microprogram address. Also shown in Figure 4 are the 'OR' and 'ZERO' inputs to the 2909. These are 8 lines which can be used to force the microprogram address to all 1's (OR) or all 0's (ZERO). These lines are used in the microcomputer for microcode routines to handle interrupts and direct memory access (DMA).

The other function in the microcomputer control loop is the macroprogram ROM. Macroprogram ROM addresses are maintained in one of the 2901 registers called the program counter. The cycle when the macroprogram ROM is addressed is called the macro-fetch cycle. The macroprogram, unlike the microprogram, will vary considerably in depth (number of words) as a function of a particular application, however, the word size is fixed at 16 bits. The macroprogram ROM shown in Figure 1 is described as N words x 16 bits. The macroprogram word is typically separated into two or three fields. The most significant 8 bits are used as a start address for a micro-sequence. The least significant 8 bits can be used as a "displacement" value which is added to the program counter to perform jumps or can be partitioned into two 4 bit fields which can be used to address the 'A' and 'B' registers in the 2901 register file.

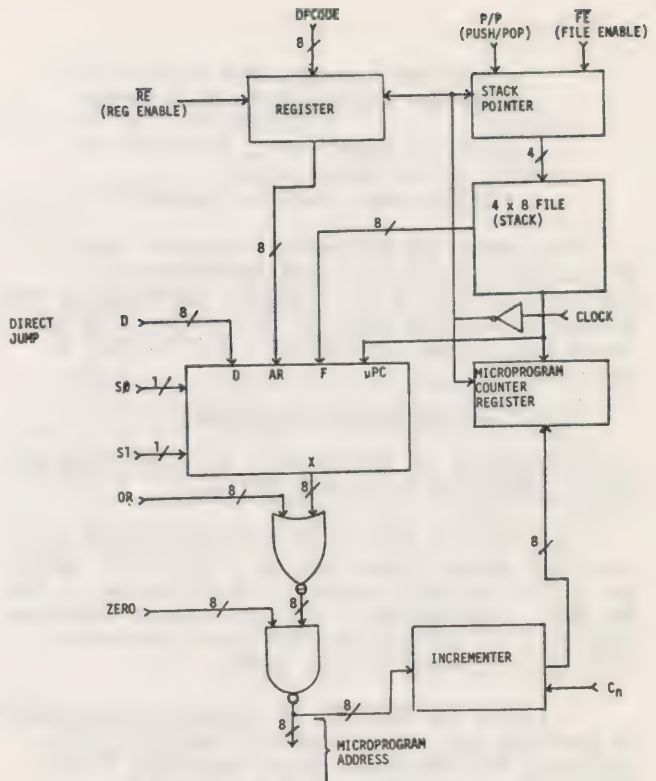


Figure 4. AM-2909 Controller - Functional Diagram

The contents of the macroprogram ROM are called macroinstructions and are equivalent to assembly language level instructions formed in general purpose computers. Each macroinstruction, therefore, contains a start address for a microinstruction sequence. The microinstruction sequence start address directs the operation performed in the microprogram and is called the opcode.

The microprogram will contain sequences of microinstructions which define macroinstructions. The application program will primarily be written in macro language, although special purpose microprogramming for special functions, I/O and interrupt structures is a feature of the bit slice microcomputer system which allows it to be tailored to a particular application.

The normal sequence of events in executing a macroinstruction is as follows:

- The program counter (an internal RALU register) contents are sent to address the macroprogram ROM.
- The address is decoded in the macroprogram ROM.
- The contents of the addressed macroprogram ROM location are loaded into the microprogram address register contained in the controller (2909). If the macroinstruction is a register to register type instruction, the A and B addresses for the RALU register file (Am, Bm) are loaded from the macroprogram ROM. This is a micro-fetch cycle.

- The microprogram controller acts as a counter and sequences through the microroutine.

- One microinstruction cycle is required to update the program counter and repeat the first event. This is the macro-fetch cycle.

Although the above sequence is rather straight forward, it becomes somewhat complicated by the fact that registers are inserted in the macro-micro control loop so that several microcycles are required from the macro-fetch cycle to the first microinstruction execution. Because of this delay, the fetching is usually running ahead of the execution, with the intermediate steps stored in the macro-micro control loop or instruction pipeline. The pipeline delay for the microcomputer system of Figure 1 is 3 microcycles. This means that it takes 3 microcycles from the macro-fetch cycle to the execution of the first microinstruction. It does not mean that each macroinstruction requires 3 microcycles, since once the pipeline is filled the pipeline delay becomes transparent. The minimum number of cycles per macroinstruction is two: one to perform the desired operation and one to update the program counter. If the desired operation requires more than one microcycle, then the micro-sequence will, of course, be longer.

Although the pipeline delay is transparent once filled for most macroinstructions, there is one class of macroinstruction that loses microcycles to the pipeline. These are microinstructions resulting in a macroprogram branch or jump.

Microcomputer Programming

Microcoding is the lowest programmable level of

control in a microcomputing system and is basically expressed in ones and zeroes. Since this level of coding is very tedious, microinstructions can also be expressed in a notation defined as register transfer language. Furthermore, a microassembler can be written to further facilitate microprogramming. The microassembler written at RCA is designed to run on a PDP-11/03 and is programmed in FORTRAN and PDP assembly language. After microcoding, macrocoding of the application program is done. This is assembly language programming and requires a macroassembler. The macroassembler also has been written for the PDP-11/03. In fact, all of the support software for the microcomputer has been modeled after PDP-11 support software.

Figure 5 shows the steps required to create the bit patterns for the microcomputer PROM's. The first step is the creation of an architecture configuration table which defines the microprogram control bits in terms of register transfer language. This table defines the microcomputer architecture and changes in the hardware can be defined at this level without affecting application software written at higher levels. The architecture configuration table is initially created using the PDP-11/03 EDIT program. This results in a disk file (SYSTEM.SRC) which reflects the hardware design. This file is used by the microassembler (MICASM) in generating a disk file (MICROM.RUN) which contains the bit patterns for the microprogram PROM. MICASM also requires another input file (MICROM.SRC). This disk file is a user input file created using the EDIT program and consists of register transfer language instructions which direct operations during each microcycle.

Most of the microcomputer application software, however, is done at a macroprogram-level. Figure 5

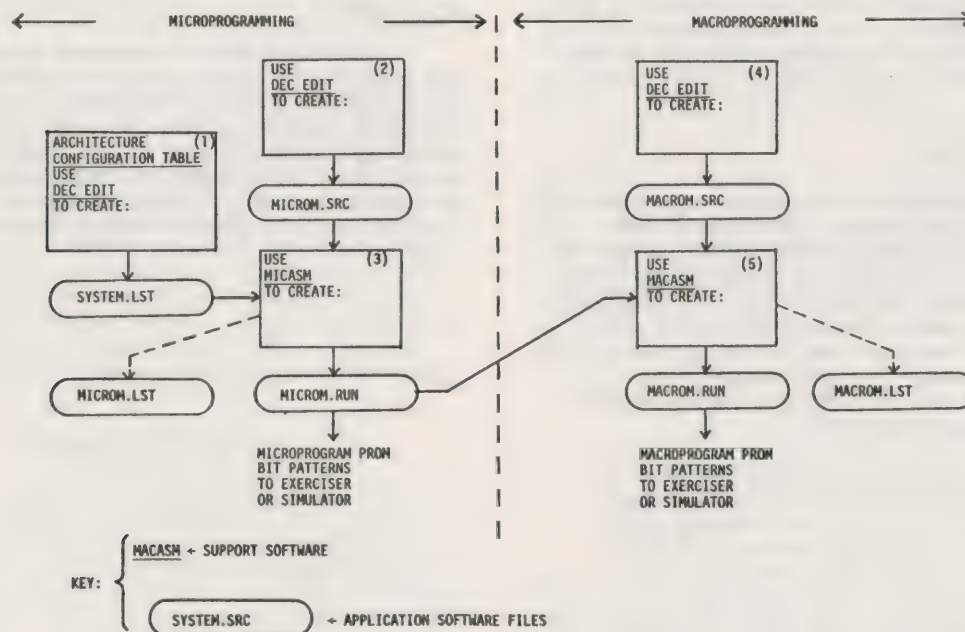


Figure 5. Microcomputer Firmware Generation

shows that the MACROM.RUN file generated by MICASM is required as an input file to the macroassembler (MACASM). The program MACASM will create a file called MACROM.RUN which contains the bit patterns for the macroprogram PROMs. The other input file required by MACASM is a user file created using the EDIT program called MACROM.SRC. This file contains assembly level language instructions for the microcomputer application macroprogram. Steps 4 and 5 in Figure 5 are the steps that a microcomputer user is normally concerned with.

Microcomputer General Purpose Instruction Set

The basic instruction set for the microcomputer has been modeled after the PDP-11/03 instruction set. This means that most instruction types used on the PDP-11/03 will run on the microcomputer. This does not mean exact emulation, since opcodes are different, however, a PDP-11/03 source program written with defined restrictions can be assembled and run on the microcomputer.

The PDP-11/03 architecture permeates the philosophy behind the microcomputer design. The program counter (PC) in the microcomputer is designated as register 7 and the stack pointer as register 6 as in the PDP-11/03. Registers 0-5 are designated as general purpose registers like the PDP-11/03. The 2901 microcomputer system has 16 registers and registers 8-16 have been designated for microcode. These registers can be accessed in macrocode, however, this is not generally recommended since execution of a special purpose microcoded function could destroy data contained in these registers.

Much of the power in this bit-slice microcomputing approach lies in the capability of implementing special purpose microprograms. Typically these include multiply (4.75 μ sec), divide (7.25 μ sec), sine/cosine (38 μ sec), and square root (38 μ sec). Additional microprograms such as FFT radix-2 and digital filters are currently being developed.

Typical Applications

In a radar system, microcomputer applications fall into three categories: off-loading main computers by processing well-defined functions in special purpose microcomputing systems; replacing minicomputers where the general purpose nature of the minicomputer provides more processing power (and cost) than required; and hardware reduction by using microprocessors to perform functions traditionally done with hardware. A few specific examples are summarized here:

- Coordinate Conversion - A basic computation task in a radar system where the antenna is located on a moving platform is to convert the variable platform coordinates to a fixed reference coordinate system. This task can be delegated to one of the system computers. Since the coordinate conversion must be updated at frequent intervals (i.e., 5 msec), and the computations involve

generating sines and cosines, considerable CPU time is consumed. Furthermore, operation with large word size (i.e., > 20 bits) is required. If this task can be removed from the central computer and done with a bit-slice microprocessor, the central computer loading can be reduced significantly.

- Data Formatting - This application provides the interface between various sensors in a radar system and a central computer. The function of this processor is to gather asynchronous data from the various sensors, place the data in standard formats, and upon command transfer the formatted data to the central computer.

- Signal Processing - Certain tracking radars (rotating antenna) require low data rate signal processing so that the antenna and range servo loops can be implemented digitally in microcomputers.

- Radar Track Loops - Complex phased-array radars track many targets simultaneously. A distributed microprocessor network is being studied to provide this function.

- Beam Steering Control - Generation of the phase tapers for a phased array is a complex real-time process. This type of processing falls within the capabilities of bit slice microcomputing architectures.

- Checkout and Monitoring - This application involves embedding microcomputers into hardware designs to provide fault detection and isolation thus simplifying field operation and maintenance.

Conclusion

Although bit-slice processors represent a more complicated approach than word or byte oriented microprocessors, they offer a great deal of processing capability and flexibility. In radar system applications where high production volumes are not usual but processing requirements are stringent, the bit-slice microprocessor in many cases is the only viable microprocessor approach.

IMCS - INTELLIGENT MOTION CONTROL SYSTEM

R. A. Patterson

E. I. Du Pont de Nemours & Company, Inc.
Engineering Physics Laboratory
Wilmington, Delaware 19898

IMCS is a microcomputer based system that drives a DC motor through complex velocity profiles specified by a few simple commands. Hardware consists of a microcomputer with a D/A converter for velocity output and a digital encoder for position feedback. The software interfaces with a terminal, solves the equations of motion, and controls motor velocity and position. System bandwidth is currently 50 HZ and profiles can be any possible combination of constant velocity, constant acceleration, and ramped acceleration segments.

Controlled motion is a major element in most industrial systems. The moving parts, linkages, motors, and controls often represent the largest system cost. They also consume a major part of the development budget. The Engineering Physics Laboratory at Du Pont is developing a system, the IMCS, that is aimed at reducing both the development and the installed costs of moving systems. IMCS uses a microcomputer and sophisticated software to drive a DC motor through complex velocity profiles that are specified with a few simple commands. It is designed to replace cams, gears, and linkages with a motor that generates the desired motion directly. In many cases the user merely has to couple the motor to his system, plug a computer terminal into the IMCS, and type in a few commands. He can immediately generate velocity profiles with constant, ramped, and cycloidal portions, and change them at will. Once the desired profiles are accomplished, they can be programmed into the IMCS and the terminal disconnected.

Hardware is simple

The IMCS hardware is quite simple as shown in Figure 1. The essential parts are a microcomputer, a 12 bit digital to analog converter, and an optional digital counter depending on the type of encoder used. There are several optional inputs and outputs for switches, lights, solenoids, etc. The serial line allows a terminal or another computer to change the velocity profile.

The hardware is a velocity output, position feedback system. The D/A outputs a velocity signal to a standard servo amplifier. The encoder feeds back motor position. The microcomputer calculates what velocity and position should be 100 times a second. If there is an error in the encoder position, it makes a correction in the velocity output to reduce the error. Normally the position error will be quite small because the control is essentially continuous within a 50 hertz bandwidth.

Commands specify profile

Figure 2 is an example of a feasible IMCS profile. It is built up from segments called primitive moves. There are three types of primitive moves, constant velocity, constant acceleration, and ramp acceleration. The user can type in the type, magnitude, and length of each primitive in the complete move. The length is actually specified as a test type and value. There are five test types. A primitive can run until it reaches a particular position, velocity, acceleration, or time, or until a particular control input changes. The IMCS then goes on to the next primitive.

The move in Figure 2 is made up of eight primitives. P1 is a ramp acceleration that goes until it reaches acceleration value a1. P2, a constant acceleration, runs until time t2. P3 runs until the velocity equals v3. The commands that specify these primitives have the following form:

```
* L T RA K 25.75 T A 20.5
* L T CA A 20.50 T T 2.20
* L T RA K -20.0 T V 70.00
etc.
* G
```

The IMCS computer types the underlined characters and the user responds. The first L means load a command. If a G were typed instead, the IMCS would execute the move. The user then enters the type and magnitude of the primitive and

then the test. The magnitudes are in terms of revolutions and seconds. There are some other commands for setting control outputs, and editing, testing, and replaying moves.

Software solves motion equations

The software consists of three tasks that run asynchronously and pass move data through FIFO buffers as shown in Figure 3. The Move Constructor is the most complicated task. In addition to solving the equations of motion, it applies the tests to determine when to switch to the next move in the Primitives List. Doing all this 100 times a second with a limited speed microcomputer requires some special tricks. For example, the equations of motion for ramp acceleration, the most complicated primitive, are:

$$a = kt$$

$$v = v_0 + kt^2/2$$

$$x = x_0 + v_0 t + kt^3/6$$

There are six multiplications and divisions to do here which could never be accomplished in 10 ms (100 times a second) on the typical microcomputer. Instead, the integral equations of motion are solved using numerical integration methods which require only addition. The integral equations of motion are:

$$a = kt$$

$$v = \int a \, dt$$

$$x = \int v \, dt$$

Applying Simpsons rule for numerical integration and some normalizations, these equations become:

$$A_i = A_{i-1} + K$$

$$V_i = V_{i-1} + A_{i-1} + A_i$$

$$X_i = X_{i-1} + V_{i-1} + 4V_{i-1} + V_i$$

The subscripts refer to values at consecutive time steps. These equations can be solved quite quickly using fixed point addition which microcomputers do very well. Very high precision is used to avoid accumulating errors in the integration. IMCS uses 32 bit numbers. Even with the overhead needed to handle 32 bit numbers in an 8 bit microcomputer, the software still has plenty of time to solve the equations 100 times a second.

Future directions

The development of the IMCS to this point is nearly complete. A lab prototype is working. However, the software is designed with future expansion in mind should the need arise. One likely improvement would be to expand it to multiple axes. The controller part of the software could be enhanced. Currently it is a simple first order transfer function. It would also be useful to add a software communications interface so IMCS could be part of a distributed control network with commands coming from a higher computer.

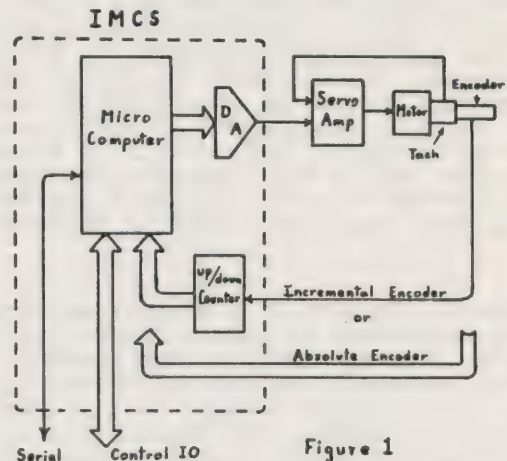


Figure 1

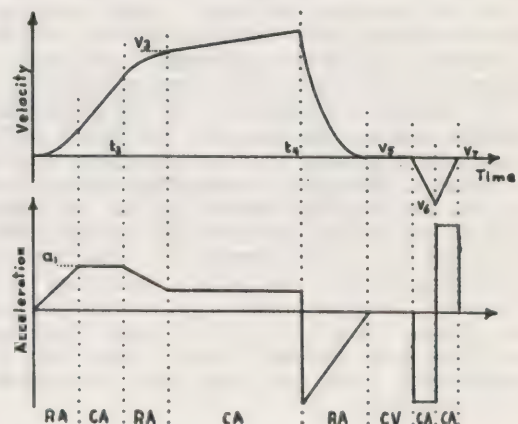


Figure 2

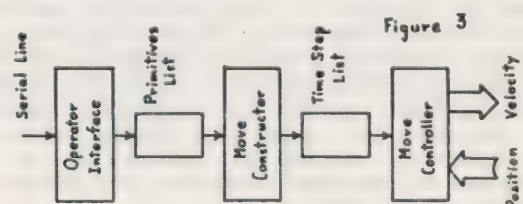


Figure 3

DATA-LINKED BAR CODE READER FOR DOCUMENTED MANUFACTURE OF HEALTH-SENSITIVE PRODUCTS

C. Leber, J. Meli, M. Daddazio, R. Jefferis

Widener College
Chester, Pennsylvania 19103

Abstract

Recent U.S.F.D.A. guidelines for good manufacturing practice of chemical compounds affecting public health require a high degree of documentation during production. Effective implementation of these guidelines under simple operating conditions, high reliability, and low cost has lead the authors to consider the use of data-linked bar code readers for production control. The paper describes the design of such a system using available micro-computer components which are capable of storing test data upon process raw materials, documenting, and recording the use of these materials, and signalling the acceptance or rejection of materials at any stage of production.

I. Introduction

Due to increasingly stringent regulations governing the manufacture of health sensitive products, a need has developed for a system to completely document the flow of materials from the reception and testing of raw materials to the distribution of finished products.

The following excerpts from the good manufacturing practice code indicate the need for such a system (21CFR).

Section 211.40 (G)

Representative samples of all dosage form drugs shall be tested to determine their conformance with the specifications for the product before distribution.

Section 211.40 (H)

Procedures shall be instituted whereby review and approval of all production control records, including packaging and labeling, shall be made prior to the distribution of a batch.

Section 211.40 (I)

Records shall indicate the quantity returned, date and actual disposition of the product.

Section 211.58

Provisions for retaining complete records of all Lab data for at least two years after distribution or one year after the drugs expiration date shall be made.

To attempt to comply with the GMP code without the use of a data based computer system is extremely undesirable due to the large amount of data that must be manipulated and retained.

The material control systems currently in use are characterized by slow information retrieval and a highly dangerous failure rate. To alleviate the problem of speed and to provide highly reliable data, a system was designed using hand-held bar code readers data linked to a data-base computer.

With this system, information regarding each stage of production, from raw materials to finished products, is transmitted to the data-base computer immediately. This reduces the time that the in-process materials must wait for approval of test results. This leads to more actual process time and less time wasted on communication inefficiencies.

The system design was broken down into two parts:

- (1) The remote scanning terminal, which includes a scanning wand (Scanamatic Samark III Verifier), a signaling device

(allowing the user to determine the status of a scan, and a micro-computer (SBC 80/10) to handle signal processing and information routing.

- (2) A host computer (Intel MDS) which will store, compile and translate all product information (i.e., present status (good or bad), amounts, etc.).

II. Remote Terminals

The remote scanning terminals have important design considerations which must be taken into account before final components are selected.

- (1) In an industrial atmosphere material strength is important, as well as protection from environmental conditions (dust, moisture, etc.).
- (2) Industrial Noise: High ambient noise levels interfering with user's signaling devices. Electrical noise interfering with the system's transmission lines. Spurious noise from mechanical equipment.
- (3) Equipment Reliability (The user must be able to scan bar code only once and get a reliable reading.)
- (4) Manual Back-Up System. Scanning terminals must be able to accommodate a keyboard entry of data if the scanning wand is inoperative or if the bar code is unreadable.
- (5) In reading the information, the system must include checks and failsafes to preclude falsely encoded data from reaching the host computer.

The scanner used is manufactured by the Scanamatic Company for the specific purpose of reading bar codes. This unit can easily handle scanning speeds of from 1/2 in. per sec. to 50 in. per sec.; a 100:1 ratio. The output is interfaced to a port on the micro-computer by Schmitt trigger signal conditioning.

When a bar code is scanned, six (6) different program modules are used to read the bar code and decode it to transmittable ASCII characters. For this system, a modification

of the common UPC bar code was chosen. The ten (10) digits (standard UPC bar code width) cannot encode enough information to facilitate total inventory and status control over the product, so a fifteen (15) digit UPC bar code has been adopted for this system. The following are two ways to implement this: First, a bar code that is fifteen digits long, with two sets of middle guard bars instead of the standard one (1), or secondly, a ten digit code with a separate five digit code located alongside of it. In this case, a pause mode is used, which allows reading both codes into the SBC 80/10 consecutively, and then transmitting both codes to the host computer as one fifteen digit code.

The UPC is a complicated code made of a series of light and dark bars. Each single width bar is called a module and there are seven modules to a character. (A bar may be made of more than one module.) There are two hidden characters in the UPC bar code, one at the beginning and one at the end. These are the code identifying character and the module check character respectively. The module check character allows each bar code to be checked for reading accuracy in the micro-computer before it is sent to the local data-base computer.

Once the bar code is read by the scanning wand and stored in the SBC 80/10's memory, it is then decoded. In this step of the process, the seven modules, which make a character, are compared against a look-up table in memory. Now a single digit can be assigned to the seven module character. Presumably, at this point, an accurate reproduction of the original bar code is now in the micro's memory. The module check character is now used to make sure that the reproduction is indeed valid. If for any reason, the bar code does not check or if, when the seven module characters were being compared against the look-up table, a match was not found, then the user does not receive a "good scan" tone. It is up to the user's discretion as to whether he should scan the bar code again or use the manual entry keyboard.

Assuming that there was no fault in reading the bar code, the user is signalled that the scan was good (acknowledgement light and variable volume tone), and now the micro-computer proceeds to convert the fifteen digit code to ASCII code. This is again implemented by means of a look-up table in memory. Once the information is completely encoded into ASCII, it is then presented to an output port of the micro and transmitted to the host computer.

The remote scanning terminals (micro-computer, bar code reader, and manual entry keyboard with signalling lights and tones) are mounted in a Nema 12 enclosure with a satellite scanning wand. In this way, the terminal is

The data-base is the "system controller". The operator is supplied with a CRT screen and a manual entry keyboard. All test results and inventory updates are entered through the data-base operator. When a scanner needs data on a specific bar code, it communicates this request via the data link to the data-base computer. The data-base system returns the status of the material to the scanner operator. This data-base computer can also provide information such as inventory amounts, and testing dates, as well as the material status, back to the data base operator. The local data-base computer would interface to a plant data-base control system as shown in Fig. 1. This plant data-base is the host computer for the

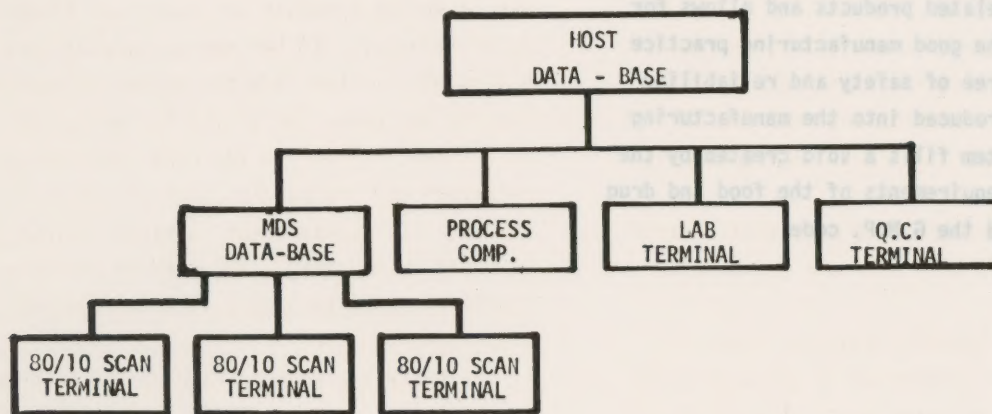


FIGURE 1 - BLOCK DIAGRAM REPRESENTATION OF COMPUTER HIERARCHY

environmentally protected and can be used in the laboratory as well as on the shipping and receiving dock.

III. Data-Base Computer

To insure a low cost and facilitate interfacing to the SBC 80/10, the data-base software was developed on the Intel "Intellec" MDS system. The ISIS II operating system was altered slightly to provide real-time capability. This was necessary since the system had to not only poll the bar code scanners, but provide for entry and retrieval of information to the data-base operator (host computer).

entire system. Test results from various labs and data from quality control are entered into the host computer from their respective terminals. This information, if applicable, is supplied to the local data-base and eventually, upon request, to the bar code readers. This hieroccybernetic structure provides an efficient means of communication between various system components. The local data-base could also communicate to process computers and lab terminals via the host computer.

All software was written using Intel's PL/M compiler language and designed using structured programming techniques with a modular building block construction. This allows for easy de-

bugging because any software problems can be readily isolated into one programming code module.

The data link, or "communications link", employs serial transmission. In this way, ASCII information is sent between the host computer and the remote terminals on a single shielded line with standard 20 MA loop current.

In a working industrial control system, only one host computer is needed and each phase of production in the plant can have its own remote terminal. This is economically feasible because of the high flexibility and low cost of the remote scanning terminals.

IV. Conclusion

This system provides a means for controlling consumer health-related products and allows for compliance with the good manufacturing practice code. A high degree of safety and reliability would thus be introduced into the manufacturing process. The system fills a void created by the upgraded safety requirements of the food and drug administration and the G.M.P. code.



FIGURE 1 - BLOCK DIAGRAM REPRESENTATION OF COMPUTER HIERARCHY

The diagram shows a hierarchical structure. At the top is the 'HOST'. Below it is the 'DATA - BASE'. The 'DATA - BASE' is connected to a 'PROCESS' block and a 'SCANNING' block. The 'SCANNING' block is further connected to three separate 'SCANNING' blocks, which represent remote scanning terminals.

